A FinOps-Integrated Platform Engineering Framework for Cost-Aware Cloud Resource Provisioning and Optimisation

Bhanu Kiran Kumar Muggalla

Platform Engineer, DevOps, Kubernetes/OpenShift, Cloud Infrastructure, AI for Infrastructure Automation Specialist
Independent Researcher, USA

Abstract

Rising cloud adoption has increased enterprise access to scalable computing, but it has also intensified uncontrolled expenditure, overprovisioning and weak cost ownership. Conventional platform-engineering practices improve delivery speed and developer self-service, yet often apply financial controls after deployment. This study designed and evaluated a FinOps-integrated platform-engineering framework that embeds cost governance within resource provisioning and continuous optimisation. A design-science method was combined with a reproducible AWS Elastic Kubernetes Service simulation comprising 30 paired runs, 40 services per run and 30 days of workload at five-minute resolution. The framework integrates an internal developer platform, pre-deployment cost estimation, budget guardrails, Infrastructure as Code templates, rightsizing, workload forecasting, automated policy enforcement and continuous cost feedback. Compared with conventional provisioning, the framework reduced mean monthly infrastructure expenditure from $2,840.53 to $1,896.08, representing a 33.25% reduction. Mean CPU utilisation increased from 23.85% to 36.20%, a gain of 12.35 percentage points, while policy violations declined from 63.60 to 9.67 per 160 provisioning requests, an 84.80% reduction. Mean provisioning time increased from 7.00 to 7.20 minutes, or 2.82%, remaining within the predefined ±5% equivalence margin. Simulated availability changed from 99.9957% to 99.9808%, remaining within the 0.10-percentage-point noninferiority margin. The study contributes a preventive cost-governance model, empirical simulation evidence and practical guidance for implementing financially accountable cloud platforms without materially reducing provisioning efficiency or reliability.

Keywords: FinOps; platform engineering; cloud cost optimisation; resource provisioning; cloud governance; Infrastructure as Code; cloud resource management

Introduction

Background of the Study

Enterprise adoption of public, private and hybrid cloud environments has increased because cloud computing provides scalable infrastructure, rapid service deployment and flexible access to computing resources. Through application programming interfaces, Infrastructure as Code and automated deployment pipelines, organisations can create servers, storage, databases and container clusters within minutes. These capabilities reduce dependence on lengthy manual infrastructure processes and support the continuous delivery of digital services (Rahman et al., 2019; X. Li et al., 2022). Automated provisioning also improves consistency because infrastructure configurations can be defined, reviewed, versioned and reused across development and production environments (Sandobalín et al., 2020).

The flexibility of cloud computing has, however, introduced financial and operational challenges. Pay-as-you-go pricing allows resources to be obtained quickly, but expenditure can grow when teams create oversized instances, retain idle services, duplicate storage, allocate unnecessary replicas or fail to terminate temporary environments. Inaccurate demand forecasts and limited visibility into resource ownership can further increase waste (Osypanka & Nawrocki, 2022; Liu et al., 2023). Although autoscaling and machine-learning methods can improve resource allocation, their effectiveness depends on workload behaviour, quality-of-service requirements and the reliability of available usage data (Garí et al., 2021; Nawrocki & Smendowski, 2025).

Platform engineering has emerged as an organisational and technical approach for managing this complexity. Platform teams develop internal developer platforms that provide approved infrastructure templates, deployment workflows, service catalogues and self-service capabilities.

These platforms reduce developers' cognitive load and create standardised pathways for delivering applications (Leite et al., 2020; Ajouaoui et al., 2023). However, most platform-engineering discussions have concentrated on developer experience, delivery speed, reliability and operational standardisation. Financial decision-making has received less direct attention, and the academic literature connecting platform engineering with cloud-cost governance remains limited (Anjum, 2026).

FinOps addresses the financial dimension of cloud operations by encouraging shared responsibility among engineering, finance, product and business teams. It promotes cost visibility, resource ownership, forecasting, allocation and continuous optimisation (Storment & Fuller, 2023). Nevertheless, FinOps activities are often performed through periodic reports, billing dashboards and post-deployment optimisation exercises. A stronger approach would integrate cost information and financial controls directly into the platform through which resources are requested and created. Cost estimation, budget validation, rightsizing recommendations and policy enforcement could then occur before deployment instead of after avoidable expenditure has already been incurred.

Statement of the Problem

Existing cloud-provisioning platforms commonly prioritise deployment speed, scalability, reliability and developer convenience. A developer may select a resource type, region, capacity and replica count without being shown the projected cost or whether the request complies with an approved budget. Consequently, financial issues may only become visible after resources have operated for several days or after the cloud bill has exceeded expectations. This makes cost management reactive rather than preventive.

Previous studies have examined cloud-cost optimisation, workload forecasting, autoscaling, Infrastructure as Code and FinOps automation separately (Lumpp et al., 2024; Nawrocki & Smendowski, 2024). However, limited research has combined these capabilities within a single platform-engineering framework that governs the entire resource lifecycle. There is therefore a need for an integrated framework that evaluates cost before provisioning, applies financial guardrails during deployment and continuously optimises resources after deployment without undermining performance or developer productivity.

Aim of the Study

This study aims to design and evaluate a FinOps-integrated platform-engineering framework for cost-aware cloud-resource provisioning and continuous optimisation.

Research Objectives

The specific objectives are to

- Examine the limitations of existing cloud-provisioning and cost-management practices.
- Design a framework that integrates FinOps capabilities into a platform-engineering environment.
- Incorporate cost estimation, budget controls, resource rightsizing and automated policy enforcement into provisioning workflows.
- Evaluate the proposed framework against conventional cloud-resource provisioning.
- Determine its effects on cloud expenditure, provisioning time, resource utilisation and governance compliance.

Research Questions

The study addresses the following questions

- How can FinOps capabilities be integrated into platform-engineering workflows?
- To what extent can the proposed framework reduce cloud-resource expenditure?
- How does cost-aware provisioning affect provisioning speed and resource utilisation?
- Can automated financial guardrails reduce budget and policy violations?
- What organisational and technical challenges affect implementation of the framework?

Research Hypotheses

The empirical evaluation will test the following hypotheses

- H1: The proposed framework significantly reduces cloud-resource expenditure compared with conventional provisioning.

- H2: The proposed framework significantly improves cloud-resource utilisation.
- H3: Automated cost controls significantly reduce budget-policy violations.
- H4: Cost-aware controls do not significantly reduce provisioning efficiency or application reliability.

Contributions of the Study

The study makes five principal contributions. First, it develops a framework connecting FinOps practices with internal developer-platform capabilities. Second, it introduces preventive cost governance by evaluating financial consequences before resources are provisioned. Third, it combines cost estimation, budget enforcement, Infrastructure as Code, rightsizing and continuous optimisation within one resource lifecycle. Fourth, it proposes an empirical evaluation method for measuring cost, utilisation, provisioning efficiency, reliability and governance compliance. Finally, it provides practical guidance for platform teams, cloud engineers, financial operations personnel and enterprise managers seeking to improve cloud accountability without restricting developer self-service.

Literature Review

Cloud Resource Provisioning

Cloud-resource provisioning involves selecting, configuring, allocating and deploying computing resources such as virtual machines, containers, storage volumes, databases and network services. Traditional provisioning relies on administrators to receive infrastructure requests, review technical requirements, configure resources and provide access to application teams. Although manual provisioning permits direct oversight, it is slow, difficult to reproduce and vulnerable to configuration errors. It also becomes unsuitable when organisations manage numerous cloud services across several development and production environments.

Automated provisioning replaces repetitive administrative tasks with scripts, templates, application programming interfaces and continuous delivery pipelines. Infrastructure as Code enables infrastructure configurations to be stored in version-controlled files and reviewed using software-development practices (Rahman et al., 2019). Tools such as Terraform, CloudFormation, Ansible and Kubernetes manifests can consistently reproduce environments and reduce the differences between approved and deployed configurations. Model-driven and code-centred provisioning approaches also improve repeatability, although their effectiveness depends on tool usability, configuration quality and integration with existing delivery processes (Sandobalín et al., 2020).

Self-service provisioning extends automation by allowing developers to request resources through catalogues, portals or approved templates without waiting for an infrastructure administrator. However, automation does not automatically guarantee efficient resource use. Inaccurate resource estimates, inappropriate instance selection, configuration drift, poor dependency management and limited visibility across cloud providers remain important challenges (X. Li et al., 2022). Infrastructure code can also contain maintainability problems, insecure defaults and inconsistent variables when appropriate standards are absent (Kumara et al., 2021). Most importantly, a technically valid request may still be financially inefficient. Provisioning workflows therefore require cost information and governance controls in addition to technical automation.

Platform Engineering

Platform engineering applies product-management and software-engineering principles to the development of reusable internal capabilities for application teams. Its central component is the internal developer platform, which provides standardised tools and services for building, testing, deploying and operating applications. Instead of requiring every development team to create separate deployment processes, a platform team maintains shared capabilities that can be consumed through application programming interfaces, command-line tools and graphical interfaces (Leite et al., 2020; Ajouaoui et al., 2023).

Developer portals provide a visible entry point to the platform. A portal may contain a

software catalogue, infrastructure templates, service documentation, ownership information, deployment status and operational metrics. By combining these capabilities, an organisation can reduce tool fragmentation and make approved infrastructure options easier to discover. Recent implementations demonstrate how self-hosted internal platforms can support agile development while retaining organisational control over infrastructure and delivery processes (Ciancarini et al., 2025).

Platform teams commonly create golden paths, which are recommended workflows for frequently performed development activities. A golden path may include an approved application architecture, continuous integration pipeline, security controls, monitoring configuration and cloud-resource template. Golden paths are intended to guide developers towards reliable practices without preventing justified exceptions. They also make self-service infrastructure safer because organisational requirements are incorporated into reusable components rather than communicated only through documentation (Dursun, 2023).

Developer experience is an important measure of platform effectiveness. A technically sophisticated platform may fail if it increases waiting time, produces unclear errors or requires developers to understand excessive infrastructure detail. Developer productivity should therefore be assessed through several dimensions, including satisfaction, performance, communication, activity and workflow efficiency (Forsgren et al., 2021). Later developer-experience research similarly emphasises feedback loops, cognitive load and the ease with which developers complete meaningful work (Forsgren et al., 2024). Current platform-engineering research remains smaller than its practitioner literature, especially regarding internal developer portals and their measurable organisational outcomes (Anjum, 2026). Cost awareness represents one such underdeveloped outcome.

FinOps Principles and Practices

FinOps is an operational approach that brings engineering, finance, product and business stakeholders together to improve the value obtained from cloud expenditure. It treats cloud cost as a shared responsibility rather than a problem assigned exclusively to the finance department. Engineers influence expenditure through architecture and resource decisions, while finance teams provide budgeting, forecasting and reporting capabilities. Business and product owners determine whether expenditure is justified by the value produced (Storment & Fuller, 2023).

Financial accountability requires each cloud resource to have an identifiable owner, purpose, environment and cost centre. Cost visibility provides stakeholders with timely information about expenditure, commitments, utilisation and variance from budget. Allocation associates shared and direct cloud charges with applications, teams, departments or customers. Accurate allocation supports showback, where teams are informed about the costs they generate, and chargeback, where those costs are transferred to departmental or product budgets. These mechanisms can influence behaviour by connecting engineering decisions with their financial consequences.

Forecasting estimates future expenditure using historical consumption, planned deployments and expected workload changes. Budgeting converts these forecasts into financial limits or targets. However, static budgets may become inaccurate when demand changes rapidly. FinOps therefore requires regular comparison of actual and forecast expenditure, followed by corrective action. Automated data collection can improve the timeliness and consistency of this process, particularly in environments where pricing and usage information comes from multiple services (Lumpp et al., 2024). Near-real-time FinOps methods have also been examined for cloud-native streaming systems, where resource demand and expenditure can change quickly (Mileski & Gusev, 2023). Continuous optimisation completes the cycle by turning cost information into rightsizing, scheduling, purchasing and architectural decisions rather than limiting FinOps to periodic reporting.

Cloud Cost-Optimisation Techniques

Rightsizing aligns provisioned capacity with observed

workload requirements. It may involve reducing an oversized virtual machine, adjusting container requests and limits, consolidating workloads or selecting a more appropriate instance family. Machine-learning methods can assist rightsizing by identifying usage patterns and predicting future resource demand (Osypanka & Nawrocki, 2022; Nawrocki & Smendowski, 2024). Rightsizing should nevertheless consider performance variability and quality-of-service requirements because aggressive reductions can increase latency or resource exhaustion.

Autoscaling adjusts capacity in response to workload conditions. Horizontal scaling changes the number of instances or pods, while vertical scaling modifies the capacity assigned to an existing resource. Cloud autoscaling research shows that policies differ in their metrics, decision rules, timing and ability to handle uncertain demand (C. Qu et al., 2018). Kubernetes Horizontal Pod Autoscaling can respond to CPU, memory or custom metrics, but its effectiveness depends on resource requests, thresholds and workload behaviour (Taherizadeh & Grobelnik, 2020; Nguyen et al., 2020). Reinforcement-learning approaches may improve adaptation, although they introduce training, stability and explainability challenges (Garí et al., 2021).

Scheduling is useful for development, testing and demonstration resources that are not required continuously. Automatically stopping non-production resources outside approved operating periods prevents idle expenditure. For stable workloads, reserved instances, savings plans or other committed-capacity arrangements can reduce unit costs compared with continuous on-demand purchasing. These commitments require accurate forecasts because unused reservations create another form of waste (Chaisiri et al., 2012; Z. Qu et al., 2023). Spot instances provide additional savings for fault-tolerant and interruptible workloads, but they are unsuitable for components that cannot withstand capacity withdrawal.

Storage optimisation includes deleting unused volumes, moving infrequently accessed data to lower-cost tiers and applying retention policies. Cloud-storage decisions must consider access frequency, retrieval charges, durability and performance rather than storage price alone (Liu et al., 2023). Organisations should also identify orphaned disks, unused addresses, inactive load balancers, outdated snapshots and abandoned test environments. These techniques are most effective when applied continuously instead of through occasional cost-reduction exercises.

Infrastructure as Code and Policy as Code

Infrastructure as Code provides the execution layer through which platform and FinOps requirements can be implemented. Terraform modules can define approved cloud resources, while Kubernetes manifests specify container requests, limits, replicas and scaling behaviour. Automated pipelines can validate syntax, calculate a deployment plan, estimate cost and require approval before changes reach production. Version control also provides an audit trail showing who changed a configuration, what was modified and when the modification occurred (Rahman et al., 2019).

Policy as Code expresses governance requirements as machine-evaluable rules. A policy engine can verify mandatory tags, approved regions, permitted instance families, encryption settings, maximum resource sizes and estimated cost thresholds. A request that violates a policy may be rejected, returned for correction or routed for documented approval. Policies should be tested and maintained like software because conflicting or outdated rules can delay delivery. Reusable modules and clear coding conventions are also necessary to control the quality problems identified in infrastructure code (Kumara et al., 2021). Combining Infrastructure as Code with Policy as Code enables financial and technical controls to be applied consistently before resources are created.

Cloud Cost Observability

Cloud cost observability connects billing information with operational context so that organisations can explain where expenditure originates and why it changes. Tagging is a basic requirement because tags identify resource owners, applications, environments, projects and cost centres. Incomplete or inconsistent tags weaken allocation and make shared expenditure

difficult to interpret. Platforms can improve tagging quality by inserting required metadata automatically during provisioning and rejecting requests with missing ownership information.

Telemetry supplies the operational evidence needed to interpret cost. CPU, memory, storage, network and request-volume metrics show whether provisioned capacity is being used effectively. When telemetry is combined with billing data, expenditure can be attributed to a service, transaction, customer or business capability. This supports unit economics, such as cost per request, cost per active user or cost per completed business process. Unit-based measures are often more informative than total expenditure because a rising cloud bill may be acceptable when service volume or revenue is increasing at a faster rate.

Dashboards communicate expenditure, forecasts, allocation, utilisation and budget variance to technical and financial stakeholders. Their effectiveness depends on data accuracy, update frequency and the actions available to users. Automated data collection reduces the delay between resource consumption and financial reporting (Lumpp et al., 2024). Forecasting models can identify expected cost patterns, while anomaly-detection methods can highlight unusual consumption before it produces a substantial budget overrun (Nawrocki & Sus, 2022; Smendowski & Nawrocki, 2024). Real-time monitoring is particularly valuable for rapidly scaling or serverless workloads (Mileski & Gusev, 2023). The literature therefore supports a platform model in which cost observability, policy enforcement and optimisation operate throughout the provisioning lifecycle rather than after deployment.

Related Frameworks and Previous Studies

Existing research relevant to this study can be divided into four streams: FinOps frameworks, platform-engineering approaches, Infrastructure as Code research, and algorithmic cloud-resource optimisation. Each stream addresses an important aspect of cloud management, but their objectives and units of analysis differ considerably.

FinOps literature treats cloud expenditure as a shared responsibility among engineering, finance, business and product teams. The FinOps operating model presented by Storment and Fuller (2023) emphasises cost visibility, allocation, forecasting and continuous optimisation. Subsequent studies have examined specific technical capabilities, including multi-cloud cost governance (F. Li et al., 2022), automated collection of cost data (Lumpp et al., 2024), cost analysis for serverless workloads (Mileski & Gusev, 2023), and machine-learning-supported optimisation of high-performance cloud computing (Nawrocki & Smendowski, 2024). These contributions demonstrate the value of financial data and automation, but they generally concentrate on reporting, data collection, or optimisation after resources have been created.

Platform-engineering research approaches the problem from a different perspective. Platform teams provide reusable infrastructure services that developers can access through self-service interfaces and standardised workflows (Leite et al., 2020). Internal developer platforms consolidate service catalogues, deployment pipelines, templates and operational tooling to reduce cognitive load and improve developer productivity (Anjum, 2026). Ciancarini et al. (2025) further demonstrate how a self-hosted, open-source internal development platform can support agile collaboration. Nevertheless, these studies largely evaluate organisational structure, developer experience and platform functionality rather than financial governance.

Infrastructure as Code provides the automation mechanism through which technical and financial policies could be enforced. However, systematic research on IaC has primarily examined tools, adoption, testing, code quality and configuration defects (Rahman et al., 2019). Cloud-optimisation studies have produced sophisticated methods for resource reservation, workload prediction, autoscaling and rightsizing (Chaisiri et al., 2012; Osypanka & Nawrocki, 2022). These methods improve resource efficiency, but they are usually treated as separate optimisation components rather

Table 1: Comparative Analysis of Related Studies

Study	*Research focus*	*Technology*	*FinOps integration*	*Evaluation method*	*Identified limitation*
Storment and Fuller (2023)	FinOps principles, organisational responsibilities and cloud financial management	Cloud billing, allocation, forecasting, showback and optimisation practices	Direct, but primarily conceptual	Practitioner knowledge, organisational cases and industry guidance	Does not provide an experimentally evaluated platform architecture for enforcing controls during provisioning
Chaisiri et al. (2012)	Minimisation of cloud-resource provisioning cost under demand and price uncertainty	Reserved and on-demand virtual-machine provisioning	None	Stochastic optimisation and numerical experiments	Cost optimisation is not integrated with self-service platforms, IaC pipelines or organisational accountability
Rahman et al. (2019)	Classification of Infrastructure as Code research, tools and research gaps	IaC frameworks, scripts and configuration-management tools	None	Systematic mapping study	Financial policies, cost estimation and budget enforcement are not examined as IaC requirements
Leite et al. (2020)	Platform teams as an organisational structure for continuous delivery	Self-service infrastructure and continuous-delivery platforms	None	Grounded Theory study involving interviews with 27 IT professionals	Explains platform-team responsibilities without addressing cost allocation, budgets or cost-aware provisioning
Osypanka and Nawrocki (2022)	Prediction and optimisation of cloud-resource usage and cost	Machine learning and cloud-resource telemetry	Partial	Predictive-model development and experimental evaluation	Concentrates on resource prediction and cost optimisation without integrating developer workflows or financial policy gates
F. Li et al. (2022)	Multi-cloud cost analysis, optimisation and governance through SmartCMP	Multi-cloud management, cost analytics and governance policies	Direct	Platform-design and cloud-management practice demonstration	Provides limited controlled evidence concerning provisioning-time guardrails, developer experience and reliability effects
Mileski and Gusev (2023)	FinOps cost analysis for near-real-time serverless streaming	Serverless functions, event streaming and public-cloud services	Direct but workload-specific	Scenario-based cost comparison covering processing and idle periods	Focuses on one serverless workload and does not provide a general platform-engineering or lifecycle-governance framework
Lumpp et al. (2024)	Automation of FinOps data collection	Dynamic scraper generation and cloud-cost data pipelines	Direct	Prototype design and technical evaluation	Addresses the cost-data acquisition layer but not end-to-end provisioning, budget enforcement, rightsizing and decommissioning
Nawrocki and Smendowski (2024)	FinOps-driven optimisation of cloud resources for high-performance computing	Machine learning, forecasting and HPC cloud infrastructure	Direct	Data-driven machine-learning experiments	The framework is domain-specific and does not integrate financial guardrails into an internal developer platform

Ciancarini et al. (2025)	Design of a self-hosted, open-source agile internal development platform	Compositional Agile System and integrated development tools	None	System design, implementation and demonstration	Emphasises collaboration and development processes without incorporating cloud-cost observability or financial controls
Nawrocki and Smendowski (2025)	Classification of methods for optimising cloud-resource consumption	Forecasting, autoscaling, scheduling, rightsizing and related techniques	Partial	Structured survey and property-based taxonomy of more than 70 studies	Provides a broad taxonomy rather than an operational architecture integrating optimisation with provisioning workflows
Anjum (2026)	*Platform engineering, internal developer portals and developer experience*	*Developer portals, service catalogues, golden paths and self-service infrastructure*	*Indirect*	*Multivocal literature review of 88 academic and practitioner sources*	*Cloud financial governance is not a central evaluation dimension, and empirical evidence concerning cost outcomes remains limited*

than capabilities embedded within a developer-facing provisioning platform.

Table 1 compares representative studies from these research streams. The limitations reported in the final column are assessed relative to the objective of the present study and should not necessarily be interpreted as limitations explicitly acknowledged by the original authors.

The comparison indicates that related frameworks are complementary rather than directly interchangeable. FinOps defines the financial capabilities required to manage cloud value, platform engineering provides the delivery environment through which developers obtain infrastructure, IaC enables repeatable implementation, and optimisation research supplies algorithms for improving resource efficiency. However, the reviewed studies provide limited evidence of a unified framework in which these capabilities operate as one continuous system.

Research Gap

The literature reveals a clear integration gap between platform engineering, FinOps and cloud-resource optimisation. Previous studies have generated valuable knowledge in each area, but the areas are commonly examined as separate technical or organisational problems. Platform-engineering research concentrates on self-service, developer productivity, standardisation and reduced cognitive load, while FinOps research concentrates on cost visibility, allocation, forecasting and optimisation. Cloud-resource-management studies predominantly investigate individual techniques such as autoscaling, rightsizing, scheduling, reservation planning and machine-learning-based prediction.

A second gap concerns the timing of financial control. Many FinOps capabilities depend on billing and utilisation data generated after deployment. Consequently, organisations may identify overspending only after resources have already accumulated costs. Although provisioning research has considered cost during capacity acquisition, such models generally do not translate cost decisions into automated approval rules within developer portals, IaC pipelines or Kubernetes deployment workflows. There is therefore limited evidence on preventive mechanisms that estimate cost, validate budgets and reject non-compliant configurations before deployment.

A third gap relates to lifecycle coverage. Existing solutions frequently address one stage, such as cost-data collection, workload forecasting, resource selection or post-deployment rightsizing. Few studies connect the complete lifecycle comprising resource request, cost estimation, policy validation, provisioning, cost attribution, runtime monitoring,

optimisation and eventual decommissioning. The absence of this feedback loop also limits the ability to transform operational cost data into improved golden paths and provisioning templates.

A further gap is evident in empirical evaluation. Platform studies commonly measure functionality, organisational arrangements or developer experience, whereas optimisation studies focus on cost, prediction accuracy or resource utilisation. Limited research evaluates cost savings alongside provisioning time, policy compliance and application reliability. This combined evaluation is necessary because strict financial controls could reduce expenditure while introducing deployment delays, insufficient capacity or service degradation.

The present study addresses these gaps by proposing a FinOps-integrated platform-engineering framework that embeds cost estimation, budget validation, tagging requirements and policy-as-code controls directly into self-service provisioning. The framework also incorporates runtime cost observability, anomaly detection, automated scheduling and rightsizing recommendations throughout the resource lifecycle. Its evaluation compares conventional provisioning with cost-aware provisioning using cloud expenditure, resource utilisation, provisioning time, financial-policy violations and application reliability as outcome measures. In this manner, the study connects the organisational accountability of FinOps, the self-service capabilities of platform engineering, the repeatability of IaC and the operational intelligence of cloud-resource optimisation within a single framework.

Proposed Framework

The proposed FinOps-integrated platform engineering framework embeds financial controls into cloud-resource provisioning and operational management. Unlike conventional approaches in which cost analysis begins after deployment, the framework evaluates cost, budget and policy requirements before infrastructure is created. It then monitors deployed resources and feeds optimisation recommendations back into the provisioning process. This creates a continuous lifecycle covering resource request, cost estimation, policy validation, deployment, monitoring, optimisation and retirement.

The framework combines six architectural layers: the developer experience layer, provisioning and orchestration layer, FinOps intelligence layer, optimisation layer, governance and policy layer, and monitoring and data layer. Human oversight operates across the layers to ensure that financially or operationally significant decisions are not executed without appropriate review.

Framework Design Principles

Cost visibility

Developers, platform teams and financial stakeholders should be able to understand the financial effect of a resource before and after deployment. Each provisioning request therefore presents an estimated monthly cost, expected cost drivers and available lower-cost alternatives. After deployment, estimated costs are compared with actual expenditure. Cost data are allocated to applications, teams, environments and business units through mandatory ownership metadata. This supports the FinOps principles of visibility, allocation and shared financial responsibility (Storment & Fuller, 2023).

Automation

Financial controls should operate as part of the provisioning workflow rather than depend on periodic manual reviews. Cost estimation, budget validation, tag verification, policy evaluation and resource registration are automatically performed before deployment. Runtime actions such as non-production scheduling, idle-resource identification and low-risk rightsizing can also be automated. Automation reduces inconsistency and enables financial governance to operate at the speed required by self-service cloud environments.

Developer self-service

The framework allows developers to request infrastructure through approved golden paths, service templates, application programming interfaces and command-line interfaces. Self-service access is retained because excessive approval processes can

reduce development speed and encourage teams to bypass the platform. Cost controls are therefore applied as embedded guardrails. When a request is rejected, the platform explains the violated rule and proposes compliant alternatives. This reflects the platform-engineering objective of reducing developer cognitive load while maintaining organisational control (Leite et al., 2020; Anjum, 2026).

Accountability

Every provisioned resource must have an identifiable owner, application, cost centre, environment and budget. These attributes are attached during provisioning and maintained throughout the resource lifecycle. Dashboards provide cost information at team, product and business-unit levels, while showback or chargeback reports communicate consumption to responsible stakeholders. Accountability remains shared because developers influence resource demand, platform teams control available templates, finance teams define budgets, and business owners approve expenditure priorities.

Policy compliance

Technical and financial requirements are expressed as machine-readable policies. Examples include permitted instance sizes, mandatory tags, approved regions, budget thresholds, storage-retention rules and restrictions on expensive resource classes. Policy checks return one of four decisions: approve, warn, escalate or deny. A request within defined limits proceeds automatically, while a high-cost or exceptional request is routed for human approval. This approach maintains control without requiring every request to pass through a manual approval process.

Scalability

The architecture should function across multiple teams, accounts, clusters and cloud providers. Its components are therefore modular and accessed through standard interfaces. Pricing adapters translate provider-specific price information into a common internal model, while reusable policies can be applied at organisation, business-unit, application or environment level. Event-driven processing and asynchronous monitoring allow the framework to analyse large resource inventories without delaying routine provisioning.

Human oversight

Automated recommendations can produce unsuitable decisions when workloads have unusual performance, regulatory or availability requirements. Human approval is therefore required for high-impact actions, including the termination of production resources, major capacity reductions, long-term purchase commitments and exceptions that exceed defined budgets. Reviewers receive the financial estimate, performance evidence and policy reasoning supporting each recommendation. Every approval, rejection and exception is recorded for audit purposes.

Continuous optimisation

Cost control does not end after deployment. Runtime utilisation, spending patterns, demand forecasts and policy events are continuously analysed to detect waste and recommend corrective action. Approved changes are returned to the provisioning layer and recorded in IaC definitions to prevent configuration drift. Repeated findings can also be used to improve golden paths. For example, if a particular template repeatedly provisions excessive capacity, its default configuration can be revised for future requests.

Framework Architecture

The six-layer architecture separates user interaction, infrastructure delivery, financial analysis, optimisation, governance and telemetry while allowing data to circulate between them. FinOps is therefore not implemented as a separate reporting dashboard. It directly influences provisioning decisions and continues to govern resources during operation.

Developer Experience Layer

The developer experience layer is the framework's primary user interface. Developers select approved services from a catalogue and provide application-specific information, including the workload type, environment, expected demand, service-level requirement, ownership and budget. The interface then displays an estimated cost and alternative configurations before submission. Golden paths provide standard configurations for

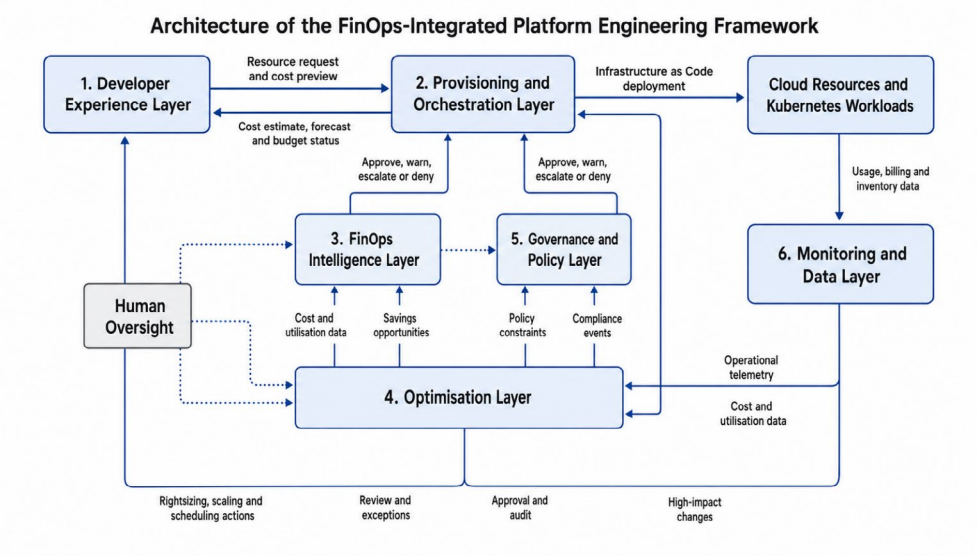

Figure 1: Proposed FinOps-integrated platform engineering architecture

common workloads such as Kubernetes applications, databases, storage services and batch-processing jobs. Although developers retain self-service access, they do not need to understand every provider-specific pricing rule or governance requirement. The layer also displays deployment status, policy results, budget consumption and optimisation recommendations.

Provisioning and Orchestration Layer

This layer converts approved requests into deployable infrastructure. It may use Terraform for cloud resources, Kubernetes manifests or Helm charts for containerised workloads, and CI/CD pipelines for workflow execution. The orchestration process contains validation, cost estimation, policy evaluation, approval and deployment stages. Infrastructure is created only when the required financial and technical checks have passed. IaC state is retained to support repeatability, change tracking and recovery. Because the desired infrastructure configuration is stored as code, approved optimisation actions can be applied consistently and reviewed through existing version-control processes (Rahman et al., 2019; Kumara et al., 2021).

FinOps Intelligence Layer

The FinOps intelligence layer supplies the financial reasoning required by the platform. It collects provider pricing, negotiated discounts, historical

Table 2: Components and outputs of the proposed framework

Layer	*Principal components*	*Main output*
Developer experience	Developer portal, service catalogue, API, CLI and golden paths	Structured and cost-visible resource request
Provisioning and orchestration	IaC engine, Kubernetes manifests, workflow engine and CI/CD pipeline	Repeatable infrastructure deployment
FinOps intelligence	Pricing engine, cost estimator, budget manager, allocation service and forecasting module	Cost estimate and financial decision
Optimisation	Rightsizing, autoscaling, scheduling, storage-tier and idle-resource analysis	Optimisation recommendation or automated action
Governance and policy	Policy engine, approval workflow, exception register and audit log	Approve, warn, escalate or deny decision
Monitoring and data	Metrics, billing, inventory, logs, telemetry and cost-attribution data	Normalised operational and financial dataset

consumption, budget data and cost-allocation rules. During provisioning, it calculates the expected cost of the requested configuration and compares it with the remaining team or project budget. It can also present alternative regions, instance families, storage tiers or purchasing models. Following deployment, the layer compares estimated and actual expenditure, supports forecasting, identifies unusual cost growth and calculates unit-cost measures such as cost per application request, customer or transaction. Automated data collection reduces the delay between resource consumption and financial analysis (Lumpp et al., 2024).

Optimisation Layer

The optimisation layer analyses cost and utilisation data to identify opportunities for reducing waste without violating application requirements. Its functions include resource rightsizing, autoscaling configuration, non-production scheduling, storage-tier optimisation, reserved-capacity recommendations, spot-instance suitability analysis and detection of idle or orphaned resources. Recommendations are prioritised according to expected savings, implementation risk and operational impact. Low-risk actions, such as shutting down an unused development resource during an approved schedule, may be automated. Changes that could affect production availability are submitted for human review. This risk-aware approach prevents cost reduction from being pursued at the expense of reliability (Osypanka & Nawrocki, 2022; Nawrocki & Smendowski, 2025).

Governance and Policy Layer

The governance and policy layer enforces organisational requirements throughout the resource lifecycle. Before deployment, it evaluates estimated cost, budget availability, ownership information, approved resource types, location restrictions and required technical controls. During operation, it checks for configuration drift, missing tags, expired exceptions and unauthorised resource changes. Policies can be inherited from organisation level and refined for particular business units or environments. An exception mechanism permits justified deviations, but each exception must include an owner, reason, approval and expiry date. The resulting audit trail provides evidence of who requested, approved, modified or terminated each resource.

Monitoring and Data Layer

The monitoring and data layer provides the operational evidence used by all other layers. It gathers billing records, cloud inventory, Kubernetes metrics, application telemetry, deployment events and policy outcomes. Common identifiers connect each cost record with its corresponding application, owner, environment and IaC deployment. Data-quality checks identify missing ownership labels, duplicate records and delayed billing information before these data are used for decision-making. Dashboards present expenditure, budget variance, utilisation, savings opportunities and policy compliance. The layer also sends events to the FinOps and governance components when abnormal expenditure, low utilisation or configuration drift is detected.

The six layers operate as a closed feedback loop. A developer begins by selecting a golden path and submitting a resource request. The provisioning layer generates the infrastructure plan, while the FinOps intelligence layer estimates its cost. The governance layer evaluates financial and technical policies. Compliant requests are provisioned automatically, while requests exceeding established thresholds are revised, escalated or denied. Once deployed, the resource is registered with the monitoring layer. Actual expenditure and utilisation are then analysed by the FinOps and optimisation layers. Approved changes are returned to the orchestration layer and recorded in the IaC configuration. This feedback process converts cost management from a periodic reporting activity into a preventive and continuously operating component of platform engineering.

Framework Components

The six architectural layers are implemented through a set of interacting components. A layer represents a broad area of responsibility, while a component performs a specific function within the provisioning and optimisation lifecycle. Together, the components convert a developer's infrastructure request into a cost-estimated, policy-compliant and continuously monitored cloud resource.

Table 3: Components of the Proposed Framework

Component	*Primary function*	*Input*	*Output*	*Expected benefit*
Developer portal	Receives provisioning requests	Resource requirements	Standardised request	Improved self-service
Cost estimator	Estimates expected cost	Configuration and prices	Predicted cost	Cost visibility
Policy engine	Evaluates financial rules	Request and policies	Approval or rejection	Budget compliance
IaC engine	Provisions resources	Approved template	Cloud resources	Consistency
Optimisation engine	Detects inefficient resources	Usage telemetry	Optimisation action	Reduced waste
FinOps dashboard	Reports cost information	Cost and usage data	Cost reports	Accountability

Developer Portal and Service Catalogue

The developer portal provides the main interface for requesting and managing cloud resources. It presents an approved service catalogue containing compute, storage, database, networking and Kubernetes deployment options. Each catalogue item is associated with a golden path that defines the recommended configuration, ownership requirements, estimated cost and applicable policies.

Developers submit information such as application name, team, environment, expected workload, service-level requirement, deployment region and budget identifier. Before the request is approved, the portal displays its expected monthly cost and possible lower-cost alternatives. The portal also provides deployment status, budget consumption, policy results and optimisation recommendations. This component supports developer self-service without requiring users to understand every cloud-provider pricing rule or infrastructure configuration detail (Leite et al., 2020; Anjum, 2026).

Template and Infrastructure as Code Repository

The template repository stores approved Terraform modules, Kubernetes manifests, Helm charts and pipeline definitions. Each template includes technical settings and financial metadata, such as default instance size, permitted regions, required tags, expected utilisation range and cost category.

Templates are version-controlled so that changes can be reviewed, tested and traced. When an optimisation recommendation is approved, the corresponding IaC definition is updated to ensure that the desired configuration matches the deployed resource. This reduces configuration drift and prevents manually applied cost-saving changes from being lost during a later deployment. IaC also supports repeatability, rollback and consistent enforcement across development, testing and production environments (Rahman et al., 2019; Kumara et al., 2021).

Cost Estimation and Pricing Engine

The cost estimation engine calculates the expected financial effect of a provisioning request before deployment. It collects cloud-provider prices, negotiated discounts, expected operating hours, storage requirements, data-transfer estimates and managed-service charges. The estimated cost of a request can be expressed as:

$$\hat{C}_r = \sum_{i=1}^{n} qipiui + C_s + C_n + C_m$$

Where Cr is the estimated request cost, qi represents the number of resource units, is the unit price, is the expected duration of use, Cs is the estimated storage cost, is Cn the network-transfer cost, and Cn represents managed-service charges.

The engine can compare alternative configurations, including regions, instance families, storage tiers and purchasing models. It returns the estimated monthly cost, the primary cost drivers and the potential savings associated with each alternative. The estimate is retained after deployment so that forecast and actual expenditure can be compared.

Budget and Quota Manager

The budget manager connects provisioning requests to financial limits established for teams, projects, products and business units. It records approved

budgets, current expenditure, committed costs and pending requests. Available budget headroom is calculated as

$$H = B - (C_a + C_c + \widehat{C_r})$$

H = B – ()

Where ***H*** represents the remaining budget headroom, ***B*** is the approved budget, Ca is actual expenditure, is Cc committed expenditure Ca, and is (Cr)ˆ the estimated cost of the new request.

Budget thresholds are configurable. A request may be approved when sufficient headroom exists, generate a warning when expenditure approaches a defined threshold, or require additional approval when it exceeds the remaining allocation. The component can also enforce service quotas, such as limits on high-cost instances, storage volumes or development clusters. These controls make budgets active provisioning constraints rather than retrospective financial reports.

Policy-as-Code Engine

The policy-as-code engine evaluates technical, operational and financial requirements. Policies may specify mandatory ownership tags, approved deployment regions, permitted instance families, encryption requirements, maximum estimated cost, budget availability and restrictions on public network exposure.

Policy evaluation occurs during the infrastructure-planning stage and continues after deployment. Each evaluation returns one of four decisions:

- *Approve*

The request satisfies all mandatory requirements.

- *Warn*

The request is permitted, but a potential cost or utilisation concern is identified.

- *Escalate*

Human approval is required because the request exceeds a financial or operational threshold.

- *Deny*

The request violates a mandatory policy and cannot proceed in its current form.

Rejected requests include an explanation of the failed rule and, where possible, a compliant alternative. Approved exceptions must contain a justification, owner, reviewer and expiry date. This prevents temporary exceptions from becoming permanent governance weaknesses.

Provisioning and Orchestration Engine

The provisioning and orchestration engine executes approved infrastructure plans. It coordinates IaC tools, Kubernetes deployments, CI/CD pipelines and cloud-provider interfaces. Its workflow consists of request validation, infrastructure planning, cost estimation, policy evaluation, approval, deployment and resource registration.

Only requests that pass the required financial and technical controls proceed to deployment. The engine maintains deployment state, records configuration changes and supports rollback when provisioning fails. Once a resource is created, its identifier and ownership information are sent to the monitoring and allocation components. This creates a traceable connection between the original request, its IaC definition, policy decisions, deployed resource and resulting expenditure.

Cost Allocation and Ownership Registry

The ownership registry maintains the organisational context of each resource. Mandatory fields include application, product, team, environment, cost centre, business unit and responsible owner. These attributes are applied during provisioning through cloud tags, Kubernetes labels and account metadata.

The registry supports showback and chargeback by assigning expenditure to the teams or products responsible for consumption. It also enables unit-cost analysis, such as cost per transaction, customer, deployment or application request. Resources without valid ownership information are flagged as non-compliant. This prevents cloud expenditure from being grouped into unidentified or shared categories that cannot be managed effectively (Storment & Fuller, 2023).

Monitoring and Cost-Observability Component

The monitoring component collects infrastructure metrics, Kubernetes telemetry, cloud inventory, billing records, logs and deployment events. These data are normalised and linked through common resource identifiers. The component tracks CPU and

memory utilisation, storage consumption, network activity, availability, actual cost, budget variance and policy compliance.

Dashboards present financial and operational information at resource, application, team and business-unit levels. Automated data collection reduces the delay between cloud consumption and financial analysis (Lumpp et al., 2024). Data-quality checks identify missing tags, delayed billing records, duplicate resources and inconsistent ownership information before the data are used for forecasting or optimisation.

Optimisation and Recommendation Engine

The optimisation engine analyses utilisation, expenditure and workload behaviour to identify savings opportunities. Its functions include rightsizing, autoscaling adjustment, non-production scheduling, reserved-capacity recommendations, spot-instance assessment, storage-tier migration and removal of idle or orphaned resources.

Each recommendation contains the estimated saving, supporting evidence, operational risk and expected effect on performance. Recommendations are ranked by financial value and implementation risk. Low-risk actions may be applied automatically, while changes affecting production capacity, availability or long-term commitments require human approval. This ensures that cost reduction is balanced against reliability and service quality (Osypanka & Nawrocki, 2022; Nawrocki & Smendowski, 2025).

Approval, Audit and Feedback Component

The approval and audit component manages exceptional or high-impact decisions. It records who submitted, reviewed, approved, rejected, modified or terminated a resource. The audit record includes the original cost estimate, policy results, approval justification, implemented configuration and subsequent optimisation actions.

The component also supports continuous platform improvement. Repeated policy violations, inaccurate cost estimates or recurring rightsizing recommendations are analysed to identify weaknesses in existing templates. The platform team can then revise service-catalogue defaults and golden paths. Consequently, operational experience is returned to the provisioning process, allowing future requests to begin with more cost-efficient and policy-compliant configurations.

Cost-Aware Provisioning Workflow

The cost-aware provisioning workflow converts a developer's resource request into a validated, financially acceptable and continuously monitored cloud deployment. Financial assessment is performed before infrastructure creation, while post-deployment telemetry supports continuous optimisation. The workflow consists of eight connected stages.

Resource Request Submission

The workflow begins when a developer selects an approved service from the internal developer portal or submits a request through an API or command-line interface. The request contains the intended resource type, application name, environment, region, expected workload, service-level requirement, deployment duration, team, owner, cost centre and budget identifier.

Where a golden path is selected, most technical parameters are automatically populated from an approved template. The developer only supplies workload-specific information. This reduces configuration complexity while ensuring that the request contains the financial and ownership metadata required for subsequent evaluation.

Request Validation

The platform performs structural and technical validation before calculating cost. It checks whether all mandatory fields have been supplied, the selected template exists, the requested resource type is supported and ownership information is valid. IaC syntax, resource dependencies, naming conventions and tag requirements are also examined.

An incomplete or invalid request is returned to the developer with a clear explanation of the failed validation rule. The platform may suggest a corrected value or approved template. Validation prevents inaccurate requests from reaching the pricing and deployment stages and reduces failures caused by inconsistent infrastructure definitions (Rahman et al., 2019).

Cost Estimation

After validation, the cost engine generates an expenditure estimate using the infrastructure plan. It retrieves current provider prices, expected operating hours, storage capacity, data-transfer requirements, managed-service charges and applicable organisational discounts. The estimate covers both fixed and usage-dependent charges.

The platform presents the projected monthly cost, major cost drivers and possible alternatives. For example, it may compare instance families, storage tiers, regions or purchasing models. The estimate is linked to the request and retained after deployment so that projected expenditure can be compared with actual cost. This makes financial information available while the developer can still revise the configuration without creating chargeable resources.

Financial and Technical Policy Evaluation

The estimated cost and infrastructure plan are submitted to the policy-as-code engine. Financial policies examine budget availability, forecasted expenditure, cost thresholds, purchasing restrictions and approved service classes. Technical policies verify region, instance type, encryption, networking, availability, security and compliance requirements.

The combined evaluation produces one of four outcomes:

Approve

All mandatory requirements are satisfied.

Warn

The request can proceed, but a potential cost or utilisation issue is reported.

Escalate

The request requires human review because its cost, risk or operational impact exceeds a defined threshold.

Deny

A mandatory policy has been violated, and the configuration must be changed.

Policy results are returned with an explanation and, where possible, an approved alternative. A denied request may therefore return to the validation and estimation stages after modification.

Human Approval of High-Risk Requests

Requests classified as high risk are sent to an authorised reviewer. Escalation may occur when the request exceeds the available budget, involves a costly production resource, requires a long-term purchasing commitment, requests an exception to an organisational policy or could significantly affect application availability.

The reviewer receives the cost estimate, budget position, technical configuration, policy results and justification supplied by the developer. The request can be approved, rejected or returned for modification. Any exception must have a responsible owner, written justification and expiry date. Human oversight $\tau\tau$ ensures that unusual business requirements can be supported without allowing unrestricted policy bypass.

- *The provisioning decision can be summarised as*

$$D = \begin{cases} \text{Provision automatically}, & V = 1,\ P = 1,\ R < \tau \\ \text{Request human approval}, & V = 1,\ R \geq \tau \\ \text{Return or deny request}, & V = 0 \text{ or mandatory policy fails} \end{cases}$$

Where V represents validation status, P represents policy compliance, R is the assessed risk level, and is the organisation's escalation threshold.

Resource Provisioning

Once approval is obtained, the orchestration engine executes the version-controlled IaC plan. Terraform may provision cloud resources, while Kubernetes manifests or Helm charts deploy containerised workloads. The platform records the deployed configuration, resource identifiers, ownership metadata, approved cost estimate and policy decision.

If deployment fails, the orchestration engine stops the workflow and initiates rollback or recovery procedures. Successfully created resources are registered with the inventory, monitoring, cost-allocation and governance components. This maintains traceability from the original request to the resulting cloud expenditure.

Continuous Usage and Cost Monitoring

Following deployment, the monitoring layer collects billing records, utilisation metrics, Kubernetes telemetry, inventory changes and policy events. Actual expenditure is compared with the original

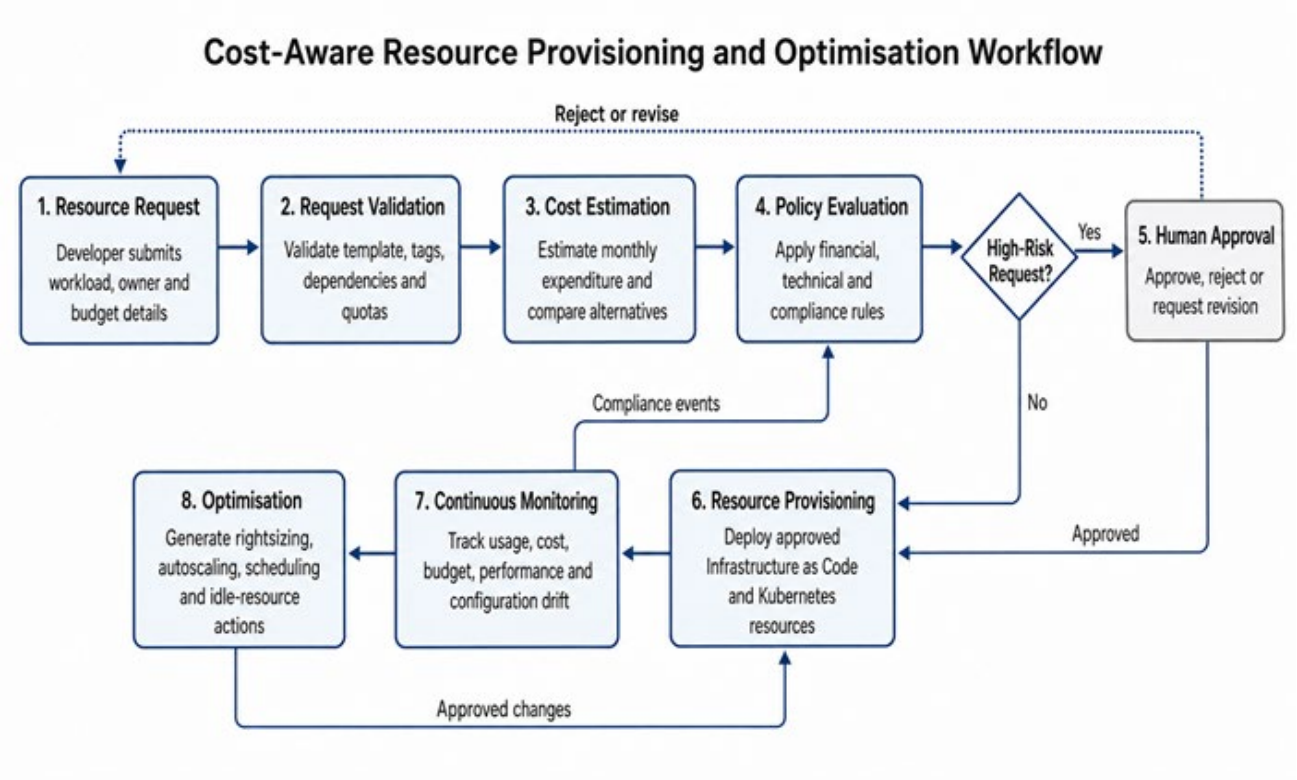

Figure 2: Cost-Aware Resource Provisioning and Optimisation Workflow

estimate and approved budget. CPU, memory, storage, network activity and availability are monitored to determine whether allocated capacity corresponds with workload demand.

Alerts are generated when actual cost exceeds its forecast, expenditure grows unexpectedly, utilisation remains below an approved threshold or ownership metadata becomes incomplete. Automated FinOps data collection reduces the delay between consumption and financial analysis (Lumpp et al., 2024). Monitoring also detects configuration drift caused by changes made outside the approved provisioning workflow.

Optimisation Recommendations and Actions

The optimisation engine evaluates monitored resources and generates actions such as rightsizing, autoscaling adjustment, shutdown scheduling, storage-tier migration, reserved-capacity purchase, spot-instance adoption and removal of idle resources. Each recommendation includes its estimated saving, supporting utilisation evidence, operational risk and expected effect on service performance.

Low-risk actions may be executed automatically when permitted by organisational policy. Examples include shutting down a development environment during an approved schedule or deleting an unattached temporary resource after its retention period. Actions affecting production capacity, reliability or long-term expenditure are submitted for human approval.

Approved changes are implemented through the orchestration engine and recorded in the IaC repository. Monitoring then continues to determine whether the action produced the expected saving without reducing reliability. This creates a closed feedback cycle in which operational evidence improves current resources and informs the design of future provisioning templates (Nawrocki & Smendowski, 2025).

Financial Governance Mechanisms

The framework applies financial governance through configurable budget thresholds. Requests approachi\ng a defined limit generate warnings, while requests exceeding the available budget require approval or revision. Mandatory tags identify the application,

Table 3: Framework Evaluation Metrics

Dimension	*Metric*	*Measurement*
Cost	Cloud expenditure	Cost per workload or month
Efficiency	Resource utilisation	Average CPU and memory utilisation
Provisioning	Provisioning time	Duration from request submission to deployment
Waste	Idle-resource rate	Percentage of provisioned resources classified as unused
Governance	Policy violations	Number of rejected or non-compliant requests
Reliability	Service availability	Uptime percentage and failed-request rate

environment, team, project, owner and cost centre associated with each resource.

Cost-centre mapping supports accurate allocation of expenditure to responsible business units. Approval limits determine which managers can authorise high-cost resources or budget exceptions. Showback reports provide teams with visibility into their consumption, while chargeback assigns actual costs to the responsible departments. Forecasting estimates future expenditure from historical and planned usage. Cost-anomaly alerts notify stakeholders when spending deviates significantly from expected patterns (Storment & Fuller, 2023).

Continuous Optimisation Mechanisms

Continuous optimisation uses operational and financial data to improve resource efficiency. Rightsizing adjusts resource capacity to match observed workload demand. Autoscaling increases or decreases capacity according to predefined performance thresholds. Idle-resource detection identifies unused instances, unattached storage and inactive services.

Scheduling shuts down non-production resources outside approved operating periods. Lifecycle management ensures that temporary resources are reviewed, renewed or removed when their retention period expires. Optimisation actions are recorded in IaC definitions to maintain consistency and prevent configuration drift (Nawrocki & Smendowski, 2025).

Human-in-the-Loop Governance

Low-risk actions may be automated within approved limits. These include scheduling development resources, correcting missing tags, adjusting non-production capacity and removing temporary resources after an approved retention period.

High-impact changes require human approval. Examples include terminating production resources, substantially reducing production capacity, purchasing long-term reserved capacity, migrating workloads between regions, exceeding budgets or overriding mandatory policies. Reviewers receive the expected savings, operational risk and supporting utilisation data. Every decision is recorded for accountability and audit purposes.

Framework Performance Indicators

The framework is evaluated using financial, operational, governance and reliability indicators. These measures allow the cost-aware approach to be compared with conventional cloud provisioning.

Research Methodology

Research Design

The study adopted a design-science research approach supported by controlled experimentation. Design science was suitable because the research involved developing and evaluating a practical framework for integrating FinOps controls into platform-engineering workflows.

The research process involved identifying deficiencies in conventional cloud provisioning, designing the proposed framework, implementing its components in a simulated AWS and Kubernetes environment, and evaluating its performance against conventional provisioning. The framework was treated as the research artefact, while cost reduction, resource efficiency, provisioning performance, governance compliance and service reliability were used as evaluation outcomes.

Thirty independent experimental replications

Table 4: Experimental Environment and Configuration

Component	*Correct experimental configuration*
Research environment	Controlled synthetic AWS EKS simulation, not a live production deployment
AWS region	us-east-1
Cluster configuration	Minimum of three worker nodes, with additional nodes allocated according to workload demand
Worker-node type	m6i.large
Node capacity	2 vCPUs and 8 GiB memory per node
Allocatable capacity	1.8 vCPUs and 7.2 GiB memory per node
EC2 price	$0.096 per m6i.large node-hour
EKS price	$0.10 per cluster-hour under standard support
Cost scope	EC2 worker-node hours and EKS control-plane fee
Excluded costs	EBS storage, data transfer, public IPv4 addresses and support-plan fees
Infrastructure as Code	Terraform, Kubernetes manifests and Helm represented in the provisioning workflow
Operational monitoring	Five-minute CPU, memory, availability and scaling telemetry
Cost monitoring	Node-hour cost calculation with OpenCost and AWS Cost Explorer-compatible reporting
Policy evaluation	OPA Gatekeeper-compatible simulation of budget, tagging and resource-profile policies
Workload portfolio	40 synthetic services comprising 28 production and 12 non-production services
Workload reporting classes	Web/API, memory-intensive, batch and non-production workloads
Provisioning requests	160 requests per experimental replication
Autoscaling configuration	50% HPA target for the baseline and 60% for the proposed framework
Maximum pod replicas	30 replicas per service
Experimental scenarios	Conventional provisioning and FinOps-integrated provisioning
Experimental repetitions	30 paired replications using identical workloads
Observation period	30 simulated days per replication
Telemetry interval	Five minutes
Statistical comparison	Paired-samples t-test or Wilcoxon signed-rank test
Significance threshold	α=0.05

were conducted for each scenario. Each replication processed 160 resource requests across four workload categories. Workload traces, pricing assumptions, traffic patterns and application requirements were held constant to ensure that observed differences resulted from the provisioning approach.

Experimental Environment

The experiment used a controlled synthetic simulation of an AWS EKS environment in the us-east-1 region. The cluster maintained a minimum of three m6i. large worker nodes, each providing 2 vCPUs and 8 GiB of memory. Additional nodes were allocated according to workload demand. Cost calculations used a fixed rate of $0.096 per EC2 node-hour and $0.10 per EKS cluster-hour. The cost model included EC2 worker-node consumption and the EKS control-plane fee. EBS storage, data transfer, public IPv4 and support-plan charges were excluded from both scenarios. Thirty paired replications were conducted, with each replication processing 160 requests over 30 simulated days.

Experimental Scenarios

Two provisioning scenarios were compared using identical workload traces and resource requirements. Scenario A represented conventional cloud-resource provisioning. Developers submitted predefined Terraform and Kubernetes configurations without receiving a pre-deployment cost estimate. Standard technical validation was applied, but there were no automated budget controls, mandatory financial tags or cost-based approval rules. Resources used fixed default requests, and cost review occurred after deployment. Rightsizing, idle-resource removal and non-production scheduling were performed reactively.

Scenario B implemented the FinOps-integrated platform-engineering framework. Every request passed through cost estimation, budget validation, mandatory tagging and policy-as-code evaluation. The framework enabled cost-aware template selection, rightsizing, autoscaling, non-production scheduling and idle-resource detection. Low-risk actions were automated, while high-cost or operationally significant changes required human approval.

Both scenarios received the same request order, workload intensity, traffic trace, application reliability requirement and random seed. This controlled arrangement allowed the financial and operational effects of the proposed framework to be isolated.

Data-Collection Procedure

Provisioning events, telemetry and cost records were collected automatically during each experimental replication. Provisioning time was measured from request submission until the workload reached a ready and available state.

The cost engine recorded the estimated monthly expenditure for every request. Actual cost was calculated from simulated resource consumption using the fixed pricing snapshot. CPU and memory utilisation were sampled every five minutes and aggregated at workload and cluster levels.

A resource was classified as idle when its average CPU utilisation remained below 5%, with negligible network activity, for at least 24 simulated hours. Policy violations included missing ownership tags, unsupported instance sizes, budget-limit breaches, prohibited regions and unapproved high-cost requests. Scaling activity was measured using the number of pod and node scaling events.

Service reliability was evaluated using application uptime and failed-request rate under identical synthetic traffic. All measurements were recorded at replication level to avoid treating repeated observations from the same run as independent samples.

Data-Analysis Method

Descriptive statistics were calculated for cloud expenditure, provisioning time, utilisation, idle-resource rate, policy violations, scaling activity and service availability. The mean, median and standard deviation were reported for each scenario.

Percentage improvement was calculated as

$$\textit{Improvement (\%)} = \frac{M_A - M_B}{M_A} \; X\,100$$

Where M_A represents the result for conventional provisioning and M_B represents the result for the FinOps-integrated framework. For indicators where a higher value was desirable, such as utilisation, the calculation was reversed.

Normality was examined at the replication level. Welch's independent-samples t-test was used when distributional assumptions were satisfied. The Mann-Whitney U test was applied when normality was not supported. Statistical significance was evaluated at $\alpha=0.05$.

Hedges' g was calculated for parametric comparisons, while rank-biserial correlation was used for non-parametric comparisons. Ninety-five percent confidence intervals were generated using 10,000 bootstrap resamples.

Provisioning time was additionally evaluated using a two one-sided tests procedure with a ±5% equivalence margin. Service availability was assessed using a 0.10 percentage-point non-inferiority margin. These tests examined whether cost controls introduced a practically important reduction in provisioning efficiency or reliability.

Validity and Reproducibility

Internal validity was supported by using identical

workloads, pricing assumptions, application configurations, traffic patterns and random seeds across both scenarios. Thirty replications reduced the influence of random workload variation. Prices remained fixed to prevent market changes from affecting scenario comparisons.

Configuration validity was maintained through version-controlled Terraform modules, Kubernetes manifests, Helm charts and policy definitions. Experimental seeds, workload traces, pricing assumptions and analysis outputs were retained to support replication.

The experiment was limited by its simulated environment. It did not fully reproduce enterprise negotiations, committed-use discounts, unpredictable human behaviour, billing-data delays or provider-level failures. Results may therefore differ in live multi-cloud or production environments. Nevertheless, the controlled design provides reproducible evidence of the relative effects of cost-aware provisioning under consistent conditions.

Results and Analysis

The results are based on 30 paired simulation replications, each representing 30 days of operation and 160 provisioning requests. All values are simulation results rather than measurements from a live AWS account. Workload-level costs were allocated using resource-consumption weights and reconcile with the total monthly expenditure.

Cloud Cost Comparison

The conventional approach produced a mean monthly cost of $2,840.53, compared with $1,896.08 under the proposed framework. This represents a saving of $944.45 or 33.25%.

The Web/API workload decreased from $926.01 to $675.00, producing a 27.11% saving. Memory-intensive workload cost declined by 28.21%, from $789.67 to $566.93. Batch workload cost decreased from $678.89 to $456.96, equivalent to a 32.69% reduction. The largest improvement occurred in non-production workloads, where cost fell from $445.96 to $197.19, a reduction of 55.78%. This result was mainly associated with off-hours scheduling, rightsizing and removal of unnecessary capacity. Worker-node consumption also decreased from 28,838.85 to 19,000.83 node-hours per month, representing a 34.11% reduction.

Provisioning-Time Comparison

Mean provisioning time increased from 7.003 minutes under conventional provisioning to 7.201 minutes under the framework. The additional financial validation, policy evaluation and telemetry registration added approximately 0.198 minutes, or 11.86 seconds, representing a 2.82% increase.

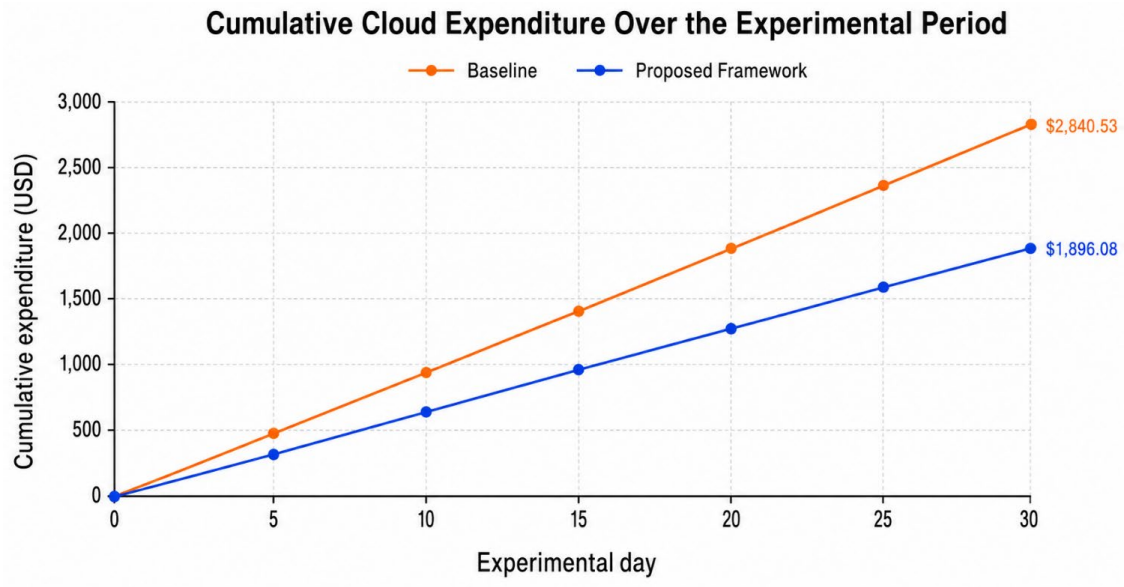

Figure 4: Cumulative cloud expenditure over the 30-day experimental period.

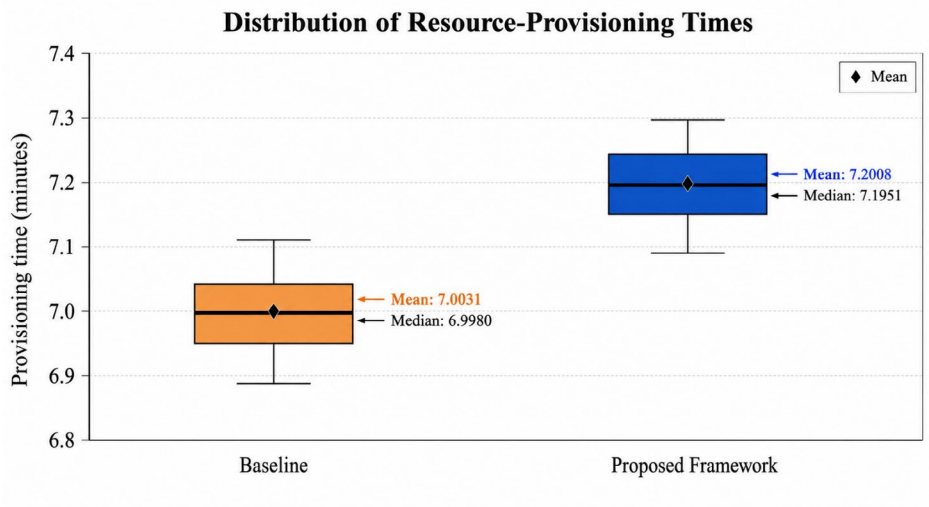

Figure 5: The distribution of resource-provisioning times

Median provisioning time was 7.008 minutes for the baseline and 7.204 minutes for the framework. Although the difference was statistically significant because of the low variation between replications, the time-ratio confidence interval of 1.0281 to 1.0283 remained within the predefined ±5% equivalence margin. Financial validation therefore introduced measurable but operationally minor overhead.

Resource-Utilisation Results

Average CPU utilisation increased from 23.85% to 36.20%. This represents an improvement of 12.35 percentage points or 51.76% relative to the baseline. Memory utilisation increased from 20.08% to 30.48%, corresponding to a 10.39 percentage-point improvement.

Storage utilisation increased from 41.23% to 57.69%, an improvement of 16.46 percentage points. These gains resulted from rightsized requests, improved workload packing, predictive scaling and better use of allocated capacity.

Service availability remained high in both scenarios. The baseline achieved 99.9957% availability, compared with 99.9808% for the framework. The difference remained within the predefined 0.10 percentage-point non-inferiority margin.

Resource-Waste Reduction

The idle-resource rate decreased from 17.33% to 5.21%, representing a 69.95% reduction. The oversized-resource rate fell from 23.44% to 6.75%, a reduction of 71.20%.

The mean number of orphaned resources per replication declined from 7.83 to 1.33, representing an 82.98% improvement. These findings indicate that scheduling, lifecycle controls, rightsizing and automated resource identification reduced the amount of provisioned capacity that produced little or no operational value.

Governance and Policy Compliance

Conventional provisioning generated an average of 63.60 policy violations per 160 requests. The proposed framework reduced this value to 9.67, representing an 84.80% reduction.

Budget violations decreased from 42.27 to 5.50 per replication, an improvement of 86.99%. Mandatory cost-allocation controls increased tagging compliance from 92.40% to 99.58%.

The framework escalated an average of 12.30 high-risk requests for human review. Of these, 4.50 were approved, while 7.80 were rejected or returned for revision. Therefore, 36.59% of escalated requests received approval and 63.41% required correction. Low-risk and compliant requests continued automatically without human intervention.

Statistical Results

Paired statistical tests were applied because identical workload seeds were evaluated under both scenarios. Cost, utilisation, waste and governance improvements

Table 5: Baseline and Proposed-Framework Results

Metric	*Baseline*	*Proposed framework*	*Percentage change*	*Statistical significance*
Web/API workload cost	$926.01	$675.00	−27.11%	p<0.001
Memory-intensive workload cost	$789.67	$566.93	−28.21%	p<0.001
Batch workload cost	$678.89	$456.96	−32.69%	p<0.001
Non-production workload cost	$445.96	$197.19	−55.78%	p<0.001
Total monthly cost	$2,840.53	$1,896.08	−33.25%	p<0.001
Mean provisioning time	7.003 min	7.201 min	+2.82%	p<0.001; equivalent within ±5%
CPU utilisation	23.85%	36.20%	+51.76%	p<0.001
Memory utilisation	20.08%	30.48%	+51.76%	p<0.001
Storage utilisation	41.23%	57.69%	+39.92%	p<0.001
Idle-resource rate	17.33%	5.21%	−69.95%	p<0.001
Oversized-resource rate	23.44%	6.75%	−71.20%	p<0.001
Orphaned resources	7.83	1.33	−82.98%	p<0.001
Policy violations	63.60	9.67	−84.80%	p<0.001
Budget violations	42.27	5.50	−86.99%	p<0.001
Tagging compliance	92.40%	99.58%	+7.78%	p<0.001
Service availability	99.9957%	99.9808%	−0.0149%	p<0.001; non-inferior

were statistically significant at p<0.001.

The mean cost difference was $944.45, with a 95% confidence interval of $917.91 to $970.99. CPU utilisation improved by 12.35 percentage points, with a 95% confidence interval of 12.23 to 12.46 percentage points. Policy violations decreased by an average of 53.93 requests, with a 95% confidence interval of 50.78 to 57.09.

Large paired effect sizes were obtained for monthly cost ($|d_z|=13.29$), CPU utilisation ($|d_z|=41.31$) and policy violations ($|d_z|=6.38$). Provisioning time satisfied the equivalence criterion, while availability satisfied the non-inferiority criterion. The results therefore supported H1, H2, H3 and H4.

Figure 5 shows a narrow distribution of provisioning times in both scenarios, indicating consistent performance across the 30 replications. The proposed framework recorded a slightly higher median provisioning time than the baseline, increasing from 6.998 to 7.195 minutes. This difference of approximately 0.20 minutes, or 12 seconds, suggests that financial and policy validation introduced only a small operational delay. The limited spread and absence of extreme values indicate that the framework maintained stable provisioning performance.

Discussion

Interpretation of Findings

The findings indicate that integrating FinOps controls into platform-engineering workflows can reduce expenditure without materially weakening provisioning performance or service reliability. Monthly cloud cost declined from $2,840.53 to $1,896.08, representing a 33.25% reduction. The largest saving occurred in non-production workloads, where scheduling and lifecycle controls reduced expenditure by 55.78%.

Resource efficiency also improved. CPU utilisation increased from 23.85% to 36.20%, while memory utilisation rose from 20.08% to 30.48%. Storage utilisation improved from 41.23% to 57.69%. These results suggest that rightsizing, improved workload packing and cost-aware autoscaling reduced the gap between allocated and consumed capacity.

The idle-resource rate decreased by 69.95%, oversized resources declined by 71.20%, and orphaned

resources fell by 82.98%. Financial governance was strengthened because policy violations declined from 63.60 to 9.67 per 160 requests. Budget violations decreased by 86.99%, while tagging compliance increased to 99.58%.

Financial validation added approximately 11.86 seconds to mean provisioning time, representing a 2.82% increase. However, this remained within the predefined ±5% equivalence margin. Availability remained within the 0.10 percentage-point non-inferiority margin. Cost visibility also improved through pre-deployment estimates, ownership tags, cost-centre mapping and workload-level reporting. However, perceived visibility among users was not directly measured.

Relationship with Previous Studies

The findings support FinOps literature that identifies cost visibility, allocation and shared responsibility as central to effective cloud financial management (Storment & Fuller, 2023). The reduction in policy violations is also consistent with SmartCMP, which demonstrates how centralised cost analysis and governance can support enterprise cloud management (F. Li et al., 2022). However, the present framework extends these approaches by applying financial controls before resources are created.

Lumpp et al. (2024) concentrated on automating FinOps data collection. The proposed framework uses similar automated data flows but connects them to budget validation, policy enforcement and continuous optimisation. Mileski and Gusev (2023) demonstrated how cloud-service selection affects serverless workload cost. The present findings extend cost-aware selection to a wider Kubernetes provisioning lifecycle.

The utilisation improvements agree with machine-learning and cloud-optimisation research showing that resource prediction, rightsizing and autoscaling can reduce unnecessary consumption (Osypanka & Nawrocki, 2022; Nawrocki & Smendowski, 2024). They also correspond with cloud-optimisation surveys that identify scheduling, scaling and idle-resource removal as important cost-control mechanisms (C. Qu et al., 2018; Nawrocki & Smendowski, 2025).

Platform-engineering studies emphasise self-service, reusable infrastructure and reduced developer cognitive load (Leite et al., 2020; Anjum, 2026). The limited provisioning overhead observed in this study suggests that financial guardrails can be added without substantially restricting developer delivery speed. IaC research has mainly examined repeatability, testing and configuration quality (Rahman et al., 2019; Kumara et al., 2021). This study extends IaC by treating cost and budget requirements as enforceable provisioning policies.

Technical Implications

The framework requires integration among developer portals, IaC pipelines, cost services, policy engines and monitoring platforms. Modular interfaces allow each component to be replaced or extended without redesigning the complete platform.

Scalability depends on asynchronous cost processing, reusable policies and hierarchical budgets. Large environments may require distributed telemetry processing and cached pricing data to prevent financial validation from delaying provisioning.

Interoperability can be supported through provider adapters and common cost, resource and ownership schemas. Automation should focus on repeatable low-risk actions, while production changes remain subject to operational safeguards and approval.

Organisational Implications

The framework makes cloud cost a shared responsibility. Developers influence workload demand, platform engineers design approved templates, finance teams define budgets and allocation rules, and operations teams protect reliability. Executive management is responsible for funding the platform and establishing acceptable cost-performance priorities.

Clear ownership is essential. Each resource should be connected to a team, project, cost centre and accountable manager. Regular reviews among engineering, finance and business stakeholders can convert cost reports into operational decisions.

Trade-Offs and Risks

Cost reduction must not compromise performance. Aggressive rightsizing or shutdown policies may

reduce capacity below workload requirements. Performance thresholds and service-level objectives should therefore constrain optimisation decisions.

Automation improves consistency but can execute incorrect recommendations at scale. Human approval remains necessary for production termination, major capacity reduction, regional migration and long-term purchasing commitments.

Financial guardrails may also restrict developer freedom. This can be reduced by providing approved alternatives and clear explanations instead of rejecting requests without guidance. Cloud-price volatility can reduce forecasting accuracy, requiring regular price updates and sensitivity analysis.

Provider-specific pricing and monitoring services may create vendor lock-in. Common data models and portable IaC templates can reduce this risk. Finally, cost telemetry may reveal sensitive business information, including customer activity and product performance. Access control, encryption, data minimisation and retention policies are therefore required.

Practical Implications

Financial institutions can use the framework to allocate cloud expenditure to products, branches and cost centres while maintaining audit records for high-cost provisioning decisions. Policy controls can also prevent unapproved regions, instance types and spending commitments.

Healthcare organisations can apply the framework to manage costly clinical and analytical workloads. Cost optimisation should remain subordinate to availability, privacy and patient-safety requirements. Production resources supporting clinical services should therefore require human approval before termination or major capacity reduction.

Telecommunications companies can benefit from predictive scaling and cost-aware capacity management because network demand varies across time and location. Scheduling and rightsizing can reduce unused capacity during low-traffic periods while maintaining resources for demand peaks.

Government cloud environments can use mandatory tagging, budget thresholds and showback reporting to improve public-sector accountability. Policy-as-code can enforce cost-centre ownership, data-residency rules and approved procurement limits across departments.

Software-as-a-Service companies can calculate cost per customer, transaction or subscription plan. These unit-cost measures help product teams determine whether customer growth produces sustainable financial value. Non-production scheduling can also reduce expenditure across development and testing environments.

Large enterprises operating multi-cloud platforms can apply common financial policies across providers. Provider adapters can normalise pricing, billing and resource information. This supports comparison of workload placement options and reduces dependence on separate cost-management processes. Nevertheless, multi-cloud implementation requires consistent resource naming, ownership metadata and policy interpretation across providers.

Successful adoption should begin with a limited group of workloads. Organisations can first introduce mandatory ownership tags, cost estimates and budget alerts before enabling automated rightsizing or resource termination. This phased approach allows teams to test policies, build confidence and determine appropriate approval boundaries.

Limitations and Future Research

The experiment covered 30 simulated days and 40 services. Although 30 paired replications were conducted, the duration may not capture seasonal workload changes, annual purchasing decisions or long-term resource growth. A larger workload portfolio could produce different cost and utilisation patterns.

The environment modelled one AWS EKS region. The findings may not transfer directly to Azure, Google Cloud or private-cloud platforms with different pricing and scaling behaviour. Cloud prices were fixed throughout the simulation, while actual prices, discounts and committed-use arrangements may change.

The cost scope included EC2 worker-node hours and the EKS control-plane fee. EBS storage, data transfer, public IPv4 and support-plan charges were excluded. Consequently, the reported savings should

not be interpreted as a complete enterprise cloud bill.

The study used synthetic workloads rather than production applications. Workload-level cost allocation, storage utilisation and waste indicators were generated within the controlled model. Real environments may contain unpredictable traffic, billing delays, manual configuration changes and application dependencies. The controlled design may also explain the large statistical effect sizes.

Organisational validation was limited. The experiment did not directly measure developer satisfaction, financial-team workload, policy acceptance or resistance to automated governance.

Future research should evaluate the framework through longitudinal organisational case studies. AI-based forecasting could improve demand and cost prediction, while reinforcement learning could support dynamic resource placement. Carbon-aware provisioning could combine cost, availability and environmental objectives. Other research opportunities include multi-cloud cost arbitrage, serverless-cost optimisation and adaptive purchasing strategies. Future studies should also evaluate privacy-preserving telemetry, explainable recommendations and the long-term effect of financial guardrails on developer experience.

Conclusion

This study addressed the problem of rising cloud expenditure caused by overprovisioning, idle resources, weak ownership and reactive cost management. Conventional platform-engineering systems improve self-service and deployment speed, but financial controls are frequently applied after resources have already generated costs.

A FinOps-integrated platform-engineering framework was proposed to incorporate cost estimation, budget validation, ownership tagging, policy enforcement, monitoring and continuous optimisation into the provisioning lifecycle. The framework connects developer self-service with financial accountability while retaining human review for high-impact decisions.

Controlled simulation results showed that the framework reduced monthly expenditure by 33.25%. CPU utilisation improved by 12.35 percentage points, while policy violations declined by 84.80%. Idle, oversized and orphaned resources were also substantially reduced. Financial validation increased provisioning time by only 2.82%, which remained within the predefined equivalence margin. Service availability also remained within the accepted non-inferiority threshold.

The study contributes an integrated connection between FinOps and platform engineering. It moves cloud financial governance from post-deployment reporting to preventive control at the point of provisioning. It also demonstrates how IaC, policy-as-code and operational telemetry can support continuous cost optimisation.

For organisations, the framework offers a structured method for balancing developer self-service, financial accountability and operational reliability. However, the results are based on a synthetic AWS EKS simulation and should be validated through production deployments, multi-cloud experiments and longitudinal organisational studies.

References

1. Storment, J. R., & Fuller, M. (2023). Cloud FinOps. " O'Reilly Media, Inc.".
2. Lumpp, F., Braga, D., Fummi, F., & Bombieri, N. (2024, December). Automating FinOps in cloud computing: An integrated solution for efficient data collection with dynamic scraper generation. In 2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 79-86). IEEE.
3. Mileski, D., & Gusev, M. (2023, November). Finops in cloud-native near real-time serverless streaming solutions. In 2023 31st Telecommunications Forum (TELFOR) (pp. 1-4). IEEE.
4. Nawrocki, P., & Smendowski, M. (2024). FinOps-driven optimization of cloud resource usage for high-performance computing using machine learning. Journal of Computational Science, 79, 102292.
5. Nawrocki, P., & Smendowski, M. (2025). A Survey of Cloud Resource Consumption Optimization Methods: P. Nawrocki and M. Smendowski. Journal of Grid Computing, 23(1), 5.

6. Jadala, S. K. (2024). Zero Trust Architecture and Identity Threat Detection for Securing Cloud, IoT, and Hybrid Enterprise Systems. International Journal of Humanities and Information Technology, 6(04), 141-185.
7. Li, F., Wu, G., Lu, J., Jin, M., An, H., & Lin, J. (2022, October). Smartcmp: A cloud cost optimization governance practice of smart cloud management platform. In 2022 IEEE 7th International Conference on Smart Cloud (SmartCloud) (pp. 171-176). IEEE.
8. Qu, Z., Dawande, M., & Janakiraman, G. (2024). Cloud cost optimization: Model, bounds, and asymptotics. Operations Research, 72(1), 132-150.
9. Osypanka, P., & Nawrocki, P. (2020). Resource usage cost optimization in cloud computing using machine learning. IEEE Transactions on Cloud Computing, 10(3), 2079-2089.
10. Liu, M., Pan, L., & Liu, S. (2023). Cost optimization for cloud storage from user perspectives: Recent advances, taxonomy, and survey. ACM Computing Surveys, 55(13s), 1-37.
11. Potla, R. B. (2024). Optimizing extended warehouse management for make-to-order plants: Slotting, wave picking, and yard orchestration at scale. Journal of Computer Science and Technology Studies, 6(3), 181-192.
12. Begimher, D., Leo, C., Huang, J., Gaw, P., & Zheng, B. (2026). SIR-Bench: Evaluating Investigation Depth in Security Incident Response Agents. arXiv preprint arXiv:2604.12040.
13. Jadala, S. K. (2025). Ransomware Resilience in Cloud and Enterprise Systems: Detection, Response, and Recovery Frameworks. Journal of Cyber-Physical Security and Robotics, 1(02), 63-86.
14. Li, X., Pan, L., & Liu, S. (2022). A survey of resource provisioning problem in cloud brokers. Journal of Network and Computer Applications, 203, 103384.
15. Zheng, H., Zhao, W., Yang, J., & Bouguettaya, A. (2012). QoS analysis for web service compositions with complex structures. IEEE Transactions on Services Computing, 6(3), 373-386.
16. Qu, C., Calheiros, R. N., & Buyya, R. (2018). Auto-scaling web applications in clouds: A taxonomy and survey. ACM Computing Surveys (CSUR), 51(4), 1-33.
17. Garí Núñez, Y., Monge Bosdari, D. A., Pacini Naumovich, E. R., Mateos Diaz, C. M., & Garcia Garino, C. G. (2021). Reinforcement learning-based application autoscaling in the cloud: a survey.
18. Taherizadeh, S., & Grobelnik, M. (2020). Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. Advances in Engineering Software, 140, 102734.
19. Nguyen, T. T., Yeom, Y. J., Kim, T., Park, D. H., & Kim, S. (2020). Horizontal pod autoscaling in kubernetes for elastic container orchestration. Sensors, 20(16), 4621.
20. Nawrocki, P., Grzywacz, M., & Sniezynski, B. (2021). Adaptive resource planning for cloud-based services using machine learning. Journal of Parallel and Distributed Computing, 152, 88-97.
21. Potla, R. (2023). Designing a BTP-centric integration mesh for shop-floor IoT, MES and ERP in discrete manufacturing. Journal of Artificial Intelligence, Machine Learning and Data Science, 1(2), 1-8.
22. Nawrocki, P., & Osypanka, P. (2021). Cloud resource demand prediction using machine learning in the context of qos parameters. Journal of Grid Computing, 19(2), 20.
23. Osypanka, P., & Nawrocki, P. (2022). QoS-aware cloud resource prediction for computing services. IEEE Transactions on Services Computing, 16(2), 1346-1357.
24. Nawrocki, P., & Sus, W. (2022). Anomaly detection in the context of long-term cloud resource usage planning. Knowledge and Information Systems, 64(10), 2689-2711.
25. Smendowski, M., & Nawrocki, P. (2024). Optimizing multi-time series forecasting for enhanced cloud resource utilization based on machine learning. Knowledge-Based Systems, 304, 112489.
26. Li, C., Bai, J., Chen, Y., & Luo, Y. (2020). Resource and replica management strategy for optimizing financial cost and user experience in edge cloud computing system. Information

Sciences, 516, 33-55.
27. Chen, X., Wang, H., Ma, Y., Zheng, X., & Guo, L. (2020). Self-adaptive resource allocation for cloud-based software services based on iterative QoS prediction model. Future Generation Computer Systems, 105, 287-296.
28. Garí, Y., Pacini, E., Robino, L., Mateos, C., & Monge, D. A. (2024). Online RL-based cloud autoscaling for scientific workflows: Evaluation of Q-Learning and SARSA. Future Generation Computer Systems, 157, 573-586.
29. Anjum, M. A. (2026). Platform engineering and internal developer portals: a multivocal literature review. Frontiers in Computer Science, 8, 1814498.
30. Ciancarini, P., Giancarlo, R., Grimaudo, G., Missiroli, M., & Xia, T. C. (2025). The design and realization of a self-hosted and open-source agile internal development platform. IEEE Access.
31. Dursun, H. (2023, June). Full spec software via platform engineering: Transition from bolting-on to building-in. In Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (pp. 172-175).
32. Ajouaoui, L., Fritzsche, A., Soeldner, G., & Soeldner, J. H. (2024). Platform Engineering for cloud-native organizations.
33. Leo, C., & Begimher, D. (2026). Survey of Attention Mechanisms in Encoder-Only Language Models. Available at SSRN 6508042.
34. Leite, L., Kon, F., Pinto, G., & Meirelles, P. (2020, June). Platform teams: An organizational structure for continuous delivery. In Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops (pp. 505-511).
35. Cuadra, J., Hurtado, E., Sarachaga, I., Estévez, E., Casquero, O., & Armentia, A. (2024). Enabling devops for fog applications in the smart manufacturing domain: A model-driven based platform engineering approach. Future Generation Computer Systems, 157, 360-375.
36. Forsgren, N., Storey, M. A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of Developer Productivity: There's more to it than you think. Queue, 19(1), 20-48.
37. Forsgren, N., Kalliamvakou, E., Noda, A., Greiler, M., Houck, B., & Storey, M. A. (2024). Devex in action. Communications of the ACM, 67(6), 42-51.
38. Leite, L., Rocha, C., Kon, F., Milojicic, D., & Meirelles, P. (2019). A survey of DevOps concepts and challenges. ACM computing surveys (CSUR), 52(6), 1-35.
39. Potla, R. B. (2024). A SOX/ITAR-aligned global ERP template for multi-plant manufacturers: Governance patterns and controls. J Artif Intell Mach Learn & Data Sci, 2(2), 3222-3232.
40. Rahman, A., Mahdavi-Hezaveh, R., & Williams, L. (2019). A systematic mapping study of infrastructure as code research. Information and Software Technology, 108, 65-77.
41. Kumara, I., Garriga, M., Romeu, A. U., Di Nucci, D., Palomba, F., Tamburri, D. A., & van den Heuvel, W. J. (2021). The do's and don'ts of infrastructure code: A systematic gray literature review. Information and Software Technology, 137, 106593.
42. Sandobalin, J., Insfran, E., & Abrahao, S. (2020). On the effectiveness of tools to support infrastructure as code: Model-driven versus code-centric. IEEE Access, 8, 17734-17761.