KubePACS: *Kube*rnetes Cluster Using *P*erformant, Highly *A*vailable, and *C*ost Efficient *S*pot Instances

Taeyoon Kim*
tykim7@hanyang.ac.kr
Dept. of Data Science
Hanyang University
Seoul, Republic of Korea

Kyumin Kim*
okkimok123@hanyang.ac.kr
Dept. of Data Science
Hanyang University
Seoul, Republic of Korea

Enrique Molina-Giménez
enrique.molina@urv.cat
Dept. of Computer Eng. and Math.
Universitat Rovira i Virgili
Tarragona, Spain

Pedro García-López
pedro.garcia@urv.cat
Dept. of Computer Eng. and Math.
Universitat Rovira i Virgili
Tarragona, Spain

Kyungyong Lee†
kyungyong@hanyang.ac.kr
Dept. of Data Science
Hanyang University
Seoul, Republic of Korea

ABSTRACT

Cloud users aim to minimize cost while maximizing performance by selecting the most suitable instance types for their workloads. To reduce expenses, spot instances have been widely adopted due to their steep discounts compared to on-demand pricing. However, their use introduces reliability risks due to potential interruptions and existing research has primarily focused on mitigating this trade-off from a cost or availability perspective alone. Despite the diversity in hardware capabilities among instance types, current provisioning systems tend to ignore performance variation, selecting nodes solely based on minimum resource requirements.

In this paper, we present KubePACS, a Kubernetes-native spot instance provisioning system that constructs node pools optimized for both cost and performance while guaranteeing high availability. KubePACS formulates the node selection process as a multi-objective optimization problem, incorporating real-time data such as spot prices, performance benchmarks, and availability scores, including the multi-node Spot Placement Score (SPS). It solves this problem efficiently using an Integer Linear Programming (ILP) approach guided by the Golden Section Search (GSS) algorithm to find the optimal configuration. By integrating with the Karpenter node autoscaler, KubePACS jointly optimizes instance-type selection and node scaling decisions within a standard provisioning workflow. KubePACS also adopts a novel heuristic to support workload-specific preferences by scaling performance metrics for specialized instances. Through extensive evaluation across synthetic and real-world workloads, KubePACS demonstrates on average 55.09% and up to 81.06% higher performance per dollar over state-of-the-art solutions such as Karpenter, SpotVerse, and SpotKube, which reference the spot instance price and limited availability data only.

*Both authors contributed equally to this work.
†Corresponding author

Middleware '26, Dec 14–Dec 18, 2026, Tarragona, Spain

ACM ISBN 979-8-4007-2621-7/26/11
https://doi.org/10.1145/3801927.3810468

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

spot instance, cost optimization, performance-aware, Kubernetes

ACM Reference Format:
Taeyoon Kim, Kyumin Kim, Enrique Molina-Giménez, Pedro García-López, and Kyungyong Lee. 2026. KubePACS: *Kube*rnetes Cluster Using *P*erformant, Highly *A*vailable, and *C*ost Efficient *S*pot Instances. In *Proceedings of 27th ACM International Middleware Conference (Middleware '26).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3810468

1 INTRODUCTION

Cloud computing has evolved into a paradigm that prioritizes performance optimization and cost-efficiency through fine-grained control over infrastructure composition. Modern cloud providers offer a wide variety of instance types with diverse compute capabilities, storage I/O throughput, and network bandwidth, enabling users to tailor infrastructure to workload-specific performance demands. At the same time, dynamic pricing models, such as spot instances, further incentivize cost savings by allowing users to access unused capacity at substantial discounts.

However, selecting optimal instance types is non-trivial. Users are often forced to choose instances based solely on static resource requirements, such as CPU cores and memory size, without fully accounting for differences in hardware performance across instance families. This leads to missed opportunities for performance-per-dollar optimization, especially in large-scale systems.

While spot instances offer compelling cost benefits of up to 90% cheaper than on-demand instances, their volatility due to potential interruptions presents additional challenges for reliable cluster formation. Although real-time availability metrics, such as Spot Placement Score (SPS) offered by Amazon Web Services (AWS) [28] and Microsoft Azure [4], have been introduced by cloud providers to improve visibility into reliability, current provisioning tools underutilize this information and lack the ability to integrate it with performance and price data in a unified optimization process.

Kubernetes, as the de facto standard for container orchestration, demands intelligent node provisioning mechanisms considering

application performance, cost-efficiency, and reliability when using spot instances. Existing tools such as Karpenter [12] focus primarily on satisfying resource constraints, such as the total number of pods or per-pod CPU and memory requirements, without accounting for differences in hardware performance, cost-performance tradeoffs, or the availability dynamics of multiple spot instances.

SpotKube [15], a Kubernetes node provisioning framework utilizing spot instances, aims to reduce costs by applying a genetic algorithm. However, it considers only spot instance prices and disregards performance heterogeneity and availability indicators, limiting its effectiveness in maximizing performance-per-dollar or ensuring robustness under spot volatility. SpotVerse [55] improves on availability awareness by incorporating datasets such as SPS and Interruption Frequency (IF). Nonetheless, it infers large-scale availability based solely on single-node SPS metrics, which are known to be imprecise for multi-node provisioning [10]. Moreover, it does not incorporate hardware performance benchmarks into its decision-making, which results in potentially suboptimal instance selection in terms of computational efficiency.

In this paper, we present KubePACS, a Kubernetes-native instance provisioning system that overcomes the limitations of prior approaches by integrating real-time, multi-dimensional spot instance datasets into the node selection process. KubePACS formulates node provisioning as a multi-objective optimization problem that jointly considers spot price, hardware performance, and large-scale availability metrics. To the best of our knowledge, KubePACS is the first system to jointly utilize multi-node SPS data, hardware performance benchmarks, and spot pricing within a unified optimization framework for cluster-level provisioning. To support workloads with specific I/O characteristics, such as network- or disk-intensive applications, KubePACS applies a workload-aware scaling heuristic that adjusts performance scores by leveraging on-demand price to reflect instance specialization. The system employs an ILP solver guided by a tunable cost-performance weight, and efficiently searches for the optimal trade-off using Golden Section Search (GSS) [48]. By integrating directly into the Kubernetes node autoscaler at the controller level, KubePACS bridges the gap between research prototype and production-ready deployment.

We evaluate KubePACS on a variety of synthetic and real-world workloads, benchmarking its effectiveness against state-of-the-art provisioning systems, including SpotVerse [55], SpotKube [15], and Karpenter [12], within the AWS cloud environment. In large-scale Kubernetes cluster provisioning scenarios, KubePACS achieves a 73.39% improvement in cost-performance efficiency compared to SpotVerse, while also enhancing the robustness of multi-node spot instance availability. This gain is attributed to KubePACS's integration of critical datasets, including spot pricing, hardware performance benchmarks, and multi-node-aware SPS. Furthermore, when running real-world graph analytics applications and I/O-intensive pipelines, KubePACS delivers up to 23.8% higher performance per dollar than Karpenter, with higher availability.

Our key contributions are summarized as follows.

- The first attempt to use benchmark score, spot price, and multi-node SPS dataset together to solve a multi-objective optimization problem to build a large-scale compute cluster.
- Proposing a workload-aware performance score adjustment heuristic for instances with specialized network and disk features.
- Applying a GSS to identify the optimal cost-performance tradeoff with minimal operational overhead.
- The open-source contribution for the research community.

2 SPOT INSTANCE KEY CONSIDERATIONS

Spot instances utilize excess cloud resources to offer substantial cost savings [38]. However, availability fluctuations might result in node interruptions necessitating an intelligent provisioning strategy.

2.1 Spot Instance Cost and Performance

While on-demand pricing correlates with hardware specifications (e.g., CPU cores, memory, I/O), spot instance pricing is dynamically determined by supply and demand, often decoupling price from hardware performance. Thus, selection strategies based solely on minimal cost may provision inferior hardware, degrading execution times and increasing total costs.

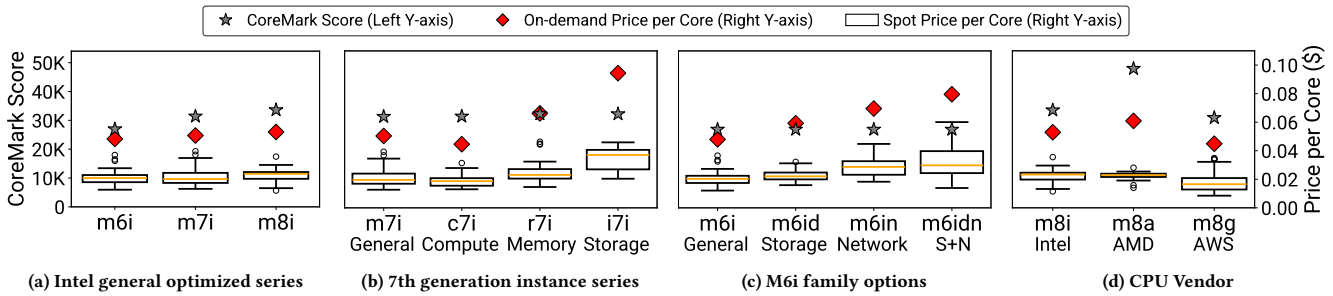
Various benchmarking tools, such as Geekbench [35], SPEC [13], and CoreMark [16], evaluate computational performance. Notably, cloud providers adopt CoreMark scores [5, 46] as a holistic measure of capability beyond raw hardware specs. However, CPU metrics alone are insufficient, as non-CPU resources such as network and disk I/O bandwidths significantly impact workload performance. Unlike CPU and memory, I/O performance exhibits substantial variability depending on configurations and usage patterns, often not scaling proportionally with allocated bandwidth. Therefore, a comprehensive evaluation incorporating multiple dimensions of instance capabilities is essential for spot instance recommendation.

Figure 1 presents CoreMark scores (gray stars) on the primary vertical axis with on-demand (red diamonds) and spot prices (boxplots [40]) on the secondary vertical axis across various AWS instance configurations. As shown in Figure 1a, newer generations (m6i to m8i) consistently yield higher computational performance, yet spot prices show a slight upward trend compared to stable on-demand costs. Figures 1b and 1c highlight that on-demand prices vary significantly based on resource optimizations (e.g., memory, storage, or networking) even when CoreMark scores remain stable, indicating that higher costs from specialized hardware do not necessarily correlate with improved compute performance. Furthermore, Figure 1d reveals that while on-demand price-to-performance ratios are consistent across CPU vendors, spot prices demonstrate distinct volatility patterns. These observations confirm that relying on spot price or CPU-centric metrics alone is insufficient for optimizing diverse workloads, as price is often decoupled from actual performance in the spot market.

2.2 Spot Instance Availability

Reliability is a crucial requirement in spot-based cluster provisioning. While early research relied on spot price volatility as a proxy for interruption risk [2, 17, 29, 29, 39, 43, 47, 58, 59], recent pricing policy changes have stabilized prices, decoupling them from real-time availability [7, 27]. Consequently, cloud providers introduced real-time availability metrics such as the SPS, offered by AWS [28] and Azure [4], ranging from 1 (Low) to 3 (High Availability).

However, existing tools often infer cluster-level availability based solely on single-node SPS [33, 38, 55], leaving a critical gap in how these metrics are utilized. To illustrate the risks, we conducted a



(a) Intel general optimized series **(b) 7th generation instance series** **(c) M6i family options** **(d) CPU Vendor**

Figure 1: Comparing benchmark score (CoreMark) and spot instance price variation. Different instance configurations show significant variations for on-demand price, hardware performance, and spot prices

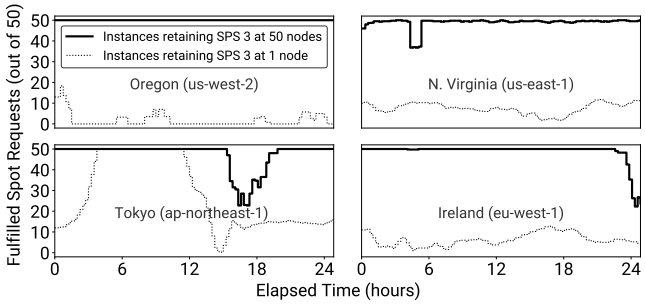


Figure 2: Different multiple spot instance availability for distinct single node and multi-node SPS values

24-hour experiment across four AWS regions, requesting 50 spot instances for two groups: one with a high SPS for 50 nodes, and another with a high SPS for a single node only.

Figure 2 empirically demonstrates the necessity of multi-node metrics. Instance types with a high multi-node SPS (solid lines) consistently maintained near-complete fulfillment (approx. 50 nodes), whereas those selected based solely on single-node SPS (dotted lines) exhibited severe provisioning failures, frequently fulfilling fewer than 10 instances despite high single-node SPS. This confirms that multi-node SPS is indispensable for large-scale spot node provisioning, a factor largely overlooked by prior approaches [43, 47, 55, 59] relying on spot prices or single-node metrics.

2.3 Kubernetes Cluster With Spot Instances

Kubernetes [50] has established itself as the standard container orchestration platform. Dynamic pod allocation via Horizontal Pod Autoscaler (HPA) [6], Cluster Autoscaler (CA), which scales worker nodes, coupled with fault-tolerance, make it suitable for batch workloads on cost-effective yet volatile spot instances.

Several systems and research projects integrate spot instances into Kubernetes. Karpenter [12] dynamically manages worker nodes by leveraging AWS SpotFleet's allocation strategies [3] for cost and availability optimization. Academic research, including SpotKube [15], SpotVerse [55], and others [8, 11, 23, 24, 54], has explored techniques such as price prediction, complete time estimation, hybrid instance selection, and migration to mitigate interruption risks.

However, state-of-the-art approaches share a critical limitation: they neglect hardware performance heterogeneity. Existing schedulers typically provision nodes based solely on static resource requirements (e.g., vCPU count and memory size), implicitly assuming uniform performance across instance types. As shown in Figure 1, this assumption leads to suboptimal selection, as instances with identical specifications can exhibit vast performance disparities. Furthermore, reliance on single-node SPS [55] does not guarantee robustness for distributed deployments, as evidenced by Figure 2. Consequently, a provisioning mechanism that jointly optimizes cost, large-scale availability, and hardware performance is required.

3 KUBEPACS: SYSTEM ARCHITECTURE

KubePACS is a framework designed to provision Kubernetes worker node pools that are **P**erformant, highly **A**vailable, and **C**ost-efficient by leveraging cloud **S**pot instances, resolving the multi-objective optimization problem of selecting optimal instance types under dynamic spot conditions. Figure 3 depicts the overall architecture composed of data ingestion, metric processing, and optimization problem solving. The workflow initiates when a user submits workload requirements, including the total number of pods, per-pod resource specifications (vCPU and memory), and workload characteristics (e.g., I/O sensitivity). Simultaneously, the *Spot Instance Data Ingestion* module aggregates real-time cloud datasets comprising hardware specifications, CPU benchmark scores (e.g., CoreMark [16]), spot prices, and multi-node availability metrics.

KubePACS processes inputs through a three-stage pipeline:

(1) **Metric Preprocessor:** Synthesizes spot price, hardware benchmark scores, and multi-node SPS to compute allocatable pod counts and scale benchmark scores per workload requirements.
(2) **Integer Linear Programming (ILP) Node Selection Solver:** Formulates a multi-objective optimization problem to select instance configurations that satisfy resource constraints while optimizing the cost-performance balance (Section 3.1).
(3) **Golden Section Search (GSS) Optimizer:** Iteratively tunes hyperparameters using the GSS algorithm [21] to identify the optimal trade-off maximizing overall efficiency (Section 3.2).

Upon identifying the optimal configuration, the system provisions the spot instances via the cloud provider's SDK, integrating them into the Kubernetes cluster as worker nodes (Section 4).

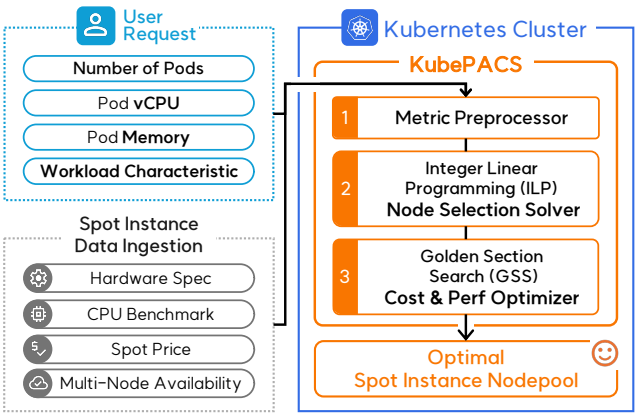


Figure 3: Overall system architecture of KubePACS

To formally formulate the resource provisioning requirements, the user's request specification is defined as Req, comprising memory (Req_{mem}), CPU cores (Req_{cpu}), and the target number of pods (Req_{pods}). Uniform-sized pods are assumed to facilitate ease-of-scaling for general workloads [18, 30, 32] on cloud. Even if multiple workloads with different pod specifications are submitted concurrently, KubePACS optimizes the node pool for each independently, enabling diverse configurations within a single cluster.

Given user preferences (e.g. instance category, region), a set of N candidate instance types is identified. Each candidate instance type, I_i ($0 \leq i < N$), represents a unique instance type within a specific AZ to account for distinct spot prices, denoted as SP_i. The allocatable CPU cores and memory of I_i are defined as CPU_i and Mem_i, respectively. To quantify computational performance, the CoreMark benchmark score [16] is adopted and denoted as BS_i.

Let x_i be the number of provisioned instances of type I_i. Number of pods allocatable to I_i, denoted as Pod_i, is derived in Equation 1 to satisfy constraints. Notations are summarized in Table 1.

$$Pod_i = \min\left(\left\lfloor \frac{CPU_i}{Req_{cpu}} \right\rfloor, \left\lfloor \frac{Mem_i}{Req_{mem}} \right\rfloor\right) \tag{1}$$

To optimize cluster composition, two efficiency metrics are introduced. First, *performance-cost efficiency* ($E_{PerfCost}$) represents the cumulative performance-per-dollar of selected instances. While maximizing $E_{PerfCost}$ promotes the selection of high-performance and cost-effective instances, doing so without constraint may cause over-provisioning by selecting more instances than necessary, increasing the overall cost. To address this issue, the *excess pod allocation efficiency* ($E_{OverPods}$) quantifies the ratio of requested pods to the total allocatable capacity, penalizing over-provisioning. Metrics are formalized in Equation 2. Given the nature of cloud, allocatable capacity is assumed to satisfy the request, i.e., $E_{OverPods} \leq 1.0$.

$$E_{PerfCost} = \sum_i \frac{BS_i \cdot x_i}{SP_i}, \quad E_{OverPods} = \frac{Req_{pod}}{\sum_i Pod_i \cdot x_i} \tag{2}$$

KubePACS aims to recommend a set of spot instances including instance types and their counts that maximizes the *total efficiency*, E_{Total}, defined as in Equation 3.

Table 1: Symbols used in the instance selection formulation

Symbol	Description
Req_{cpu}	The requested number of CPU cores per pod
Req_{mem}	The requested memory size per pod
Req_{pod}	The total number of requested pods
N	The total number of candidate instance types
I_i	A candidate instance type with index i
CPU_i	The number of CPU cores provided by I_i
Mem_i	The memory size provided by I_i
SP_i	Spot Price of I_i
OP_i	On-demand Price of I_i
BS_i	Single core Benchmark Score of I_i
Pod_i	Number of pods that can be placed on I_i
$T3_i$	Maximum number spot instances of I_i where SPS being 3
x_i	Number of I_i instances to be allocated
$Perf_i$	Total Benchmark score of I_i, given by $BS_i \cdot Pod_i$
α	A hyperparameter to balance between performance and cost

$$E_{Total} = E_{PerfCost} \times E_{OverPods} \tag{3}$$

In this formulation, $E_{PerfCost}$ encourages the use of spot instances with high benchmark scores and significant cost savings, while $E_{OverPods}$ penalizes solutions that result in excessive resource allocation beyond the workload's requirements. This combination enables the system to balance performance with cost efficiency.

3.1 Optimal Node Selection Solver Design

Assigning pods with specific resource requirements to spot instances can be formulated as a bin-packing problem [36]. However, the dynamic price and heterogeneous performance of spot instances necessitate a joint optimization of cost-efficiency and computational power, extending the problem beyond the traditional bin-packing algorithm. Consequently, this work formulates the allocation strategy as an Integer Linear Programming (ILP) problem to handle multi-dimensional constraints optimally.

The total performance contribution of an instance I_i is defined as $Perf_i = BS_i \times Pod_i$, representing the aggregate benchmark score for all hosted pods. To resolve scale discrepancies between large benchmark scores and small hourly costs when solving an ILP problem, Min-Normalization is applied to both metrics where the minimum of each metric is shown in Equation 4, chosen for its demonstrated effectiveness in multi-objective scaling [45].

$$Perf_{\min} = \min_{0 \leq i < N} (BS_i \cdot Pod_i), \quad SP_{\min} = \min_{0 \leq i < N} (SP_i) \tag{4}$$

To maximize E_{total}, the ILP solver must account for the pod over-allocation factor $E_{OverPods}$ during objective function evaluation. However, since $E_{OverPods}$ can only be computed after solving the allocation problem, it introduces a cyclic dependency that renders the objective function unsolvable using standard ILP solvers.

Directly maximizing the total efficiency E_{Total} is infeasible due to a cyclic dependency. The over-allocation factor, $E_{OverPods}$, depends on the final allocation count, which is the output of the solver itself. A naive approach of enforcing strict equality between allocated and requested pods restricts the solution space, potentially precluding configurations where slight over-provisioning utilizes cheaper, larger instances to improve overall efficiency.

To mitigate this, the objective function is formulated to determine the optimal instance count x_i by minimizing a weighted sum of normalized performance and cost, as presented in Equation 5.

$$\text{minimize} \sum_i \left(-\alpha \cdot \frac{Perf_i}{Perf_{min}} + (1-\alpha) \cdot \frac{SP_i}{SP_{min}} \right) \cdot x_i \quad \text{subject to:} \quad x_i \le T3_i, \quad x_i \in \mathbb{Z}_{\ge 0} \tag{5}$$

The formulation introduces a tunable hyperparameter $\alpha \in [0.0, 1.0]$ to balance the trade-off between maximizing performance and minimizing cost. A higher α prioritizes performance that potentially leads to the over-provisioning, while a lower α favors strict cost reduction. To guarantee robust availability of selected spot instances, the constraint $x_i \le T3_i$ leverages the multi-node SPS dataset [10, 34, 38]. Specifically, $T3_i$ is defined as the maximum number of simultaneous instances for type I_i that maintain an SPS of 3 (highest availability). Given the non-increasing nature of SPS with respect to request size [10], limiting allocations to $T3_i$ ensures that the provisioned cluster operates with high availability, thereby minimizing interruption risks.

3.2 Cost-Performance Hyperparameter Tuning

The optimal instance configuration minimizing Equation 5 is highly sensitive to the weight parameter α. Low α values prioritize cost reduction, restricting node counts to the minimum required. Conversely, high α values emphasize performance, potentially justifying over-provisioning if the performance gain outweighs the marginal cost increase. Consequently, identifying the specific α that maximizes the total efficiency metric, E_{Total}, is essential.

To identify this optimal α, we employ the golden-section search (GSS) algorithm [21], a classical method for efficiently locating the maximum of a unimodal function. GSS iteratively narrows the search interval using the golden ratio, $\phi = \frac{\sqrt{5}-1}{2} \approx 0.618$, by selecting two interior points that divide the interval proportionally. The point with the lower function value is discarded at each step, contracting the interval by a factor of approximately ϕ. GSS is chosen for its superior convergence rate compared to alternatives like ternary search. By reusing intermediate function evaluations, GSS requires only one objective function evaluation per iteration after initialization [9]. This characteristic minimizes the computational overhead of the iterative ILP solving process.

The search operates within the range $\alpha \in [0.0, 1.0]$ and terminates when the interval width falls below a tolerance $\varepsilon = 10^{-n}$. The number of iterations, k, required to guarantee this precision is derived from the contraction factor $\phi \approx 0.618$ [9]. The general bound of k is defined as follows.

$$k - 1 \ge \left\lceil \frac{\log(\varepsilon/(b-a))}{\log(\phi)} \right\rceil$$

Given $(b-a) = 1.0$ and $\varepsilon = 0.1^n$, we obtain:

$$k - 1 \ge \left\lceil \frac{\log(10^{-n})}{\log(0.618)} \right\rceil = \left\lceil \frac{-n \cdot \log(10)}{\log(0.618)} \right\rceil \approx \lceil 4.784 \cdot n \rceil \tag{6}$$

Approximately $5n + 1$ iterations are required to achieve the tolerance ε, with the search range tolerance shrinks exponentially when the number of iterations increases linearly with n. Empirical analysis (Figure 7) suggests that $n = 2$, tolerance of 0.01, achieves an optimal balance, locating the target α with negligible overhead.

3.3 Workload-Aware Performance Scaling

While CoreMark robustly measures compute-bound performance, it fails to capture the performance benefits of specialized network or storage hardware. As shown in Figure 1, cloud providers impose price increments for such features. Consequently, the optimization solver (Equation 5) would penalize these instances due to elevated costs (SP_i) and identical CPU benchmarks (BS_i), despite their suitability for I/O-bound applications.

To address this discrepancy, a scaling mechanism is applied when a user specifies workload preference (e.g., *network-* and/or *disk-intensive*). For instance types matching the requested capability, the benchmark score BS_i is adjusted using the ratio of their on-demand price to the base instance price.

$$BS_i^{scaled} = BS_i \times \frac{OP_i}{OP_{base}}$$

Here, OP_i denotes the on-demand price of specialized instance I_i, and OP_{base} the price of the corresponding general-purpose instance in the same family. This approach leverages the cloud provider's pricing model, implicitly quantifying the value of hardware enhancements where direct benchmarking is impractical.

For example, in a *network-intensive* scenario, the score of a *c6in* (network-optimized) instance is scaled up by the ratio of its price, \$0.23, to the base *c6i* of \$0.17, effectively increasing its competitiveness by $\frac{0.23}{0.17}$ in the solver. Conversely, non-matching types like *c6id* (disk-optimized) remain unscaled. These adjusted scores are subsequently integrated into the objective function to ensure appropriate resource selection.

When no workload preference is specified, the workload-aware scaling is not applied and all candidate instances are evaluated with uniform benchmark weighting. Even if an incorrect preference is provided, the system provisions a fully functional cluster, as only the hardware specialization scoring is affected without compromising availability or correctness.

Algorithm 1 outlines the procedure for generating a cost-efficient and reliable worker node configuration, comprising two stages.

In the first stage (Lines 3–6), the system aggregates and normalizes instance-level metadata. The function *DatasetPreProcessing* takes a candidate instance I_i and per-pod resource requirements ($Req_{\text{cpu}}, Req_{\text{mem}}$) as inputs. It computes the maximum allocatable pod count (Pod_i) for the instance and scales the benchmark score (BS_i) according to the user-defined workload preferences (e.g., disk and/or network heavy). The output is an enriched dataset $\mathcal{I}$ containing all necessary attributes for optimization.

In the second stage, Lines from 7, the system executes a hyperparameter optimization over the cost-performance weight $\alpha \in$

Algorithm 1 KubePACS Node Selection

Input: Pod spec($Req_{\text{pod}}, Req_{\text{cpu}}, Req_{\text{mem}}$), workload W
Output: Node pool configuration $\{(I_i, x_i)\}$ satisfying total pod requirement
Initialize $\mathcal{I} \leftarrow \emptyset$
for each instance type I_i **do**
  $\mathcal{I} \leftarrow \mathcal{I} \cup \text{DatasetPreProcessing}(I_i, W, Req_{\text{cpu}}, Req_{\text{mem}})$
end for
Initialize search interval: $\alpha_{\text{left}} \leftarrow 0.0$, $\alpha_{\text{right}} \leftarrow 1.0$
$\phi \leftarrow 0.618$ ▷ Golden ratio
$(\alpha_1, \alpha_2) \leftarrow (\alpha_{\text{right}} - \phi,\ \alpha_{\text{left}} + \phi)$
$\mathcal{S}_1 \leftarrow \texttt{ILP}(\alpha_1, \mathcal{I}, Req_{\text{pod}})$, $\mathcal{S}_2 \leftarrow \texttt{ILP}(\alpha_2, \mathcal{I}, Req_{\text{pod}})$
$\mathcal{S}^* \leftarrow$ argmax between $(\mathcal{S}_1,\ \mathcal{S}_2)$ by E_{total}
while $\alpha_{\text{right}} - \alpha_{\text{left}} > \varepsilon$ **do**
  if $E_{\text{total}}(\mathcal{S}_1) \geq E_{\text{total}}(\mathcal{S}_2)$ **then**
    $\alpha_{\text{right}} \leftarrow \alpha_2$
    $\alpha_2 \leftarrow \alpha_1, \mathcal{S}_2 \leftarrow \mathcal{S}_1$
    $\alpha_1 \leftarrow \alpha_{\text{right}} - \phi \cdot (\alpha_{\text{right}} - \alpha_{\text{left}})$
    $\mathcal{S}_1 \leftarrow \texttt{ILP}(\alpha_1, \mathcal{I}, Req_{\text{pod}})$
    $\mathcal{S}^* \leftarrow$ argmax between $(\mathcal{S}^*,\ \mathcal{S}_1)$ by E_{total}
  else
    $\alpha_{\text{left}} \leftarrow \alpha_1$
    $\alpha_1 \leftarrow \alpha_2, \mathcal{S}_1 \leftarrow \mathcal{S}_2$
    $\alpha_2 \leftarrow \alpha_{\text{left}} + \phi \cdot (\alpha_{\text{right}} - \alpha_{\text{left}})$
    $\mathcal{S}_2 \leftarrow \texttt{ILP}(\alpha_2, \mathcal{I}, Req_{\text{pod}})$
    $\mathcal{S}^* \leftarrow$ argmax between $(\mathcal{S}^*,\ \mathcal{S}_2)$ by E_{total}
  end if
end while
return Solution $\mathcal{S}^*$ with highest E_{total}

$[0.0, 1.0]$ using the GSS algorithm to maximize E_{total}. For each candidate α, the solver function *ILP* is invoked to generate a candidate node pool $\mathcal{S}$. This function formulates the selection task as an ILP problem (Equation 5), determining the optimal set of instances that satisfies the total pod demand (Req_{pod}) and availability constraints ($T3_i$). The solver aims to jointly minimize cost and over-allocation while maximizing hardware performance under the specified α. The GSS algorithm iteratively refines α, ultimately returning the configuration $\mathcal{S}^*$ that yields the highest overall efficiency among the evaluated candidates.

4 KUBEPACS IMPLEMENTATION

KubePACS is implemented as a Python-based prototype module integrated directly into the *Karpenter Controller* to facilitate Kubernetes-native node auto-scaling. The overall workflow is shown in Figure 4. The system utilizes a forked codebase of Karpenter [12], intercepting the standard node expansion workflow triggered by *Pending Pods* to inject the proposed KubePACS instance selection logic. To construct an optimal node pool that maximizes the proposed objective function, E_{total}, the problem is modeled and solved using the Python PuLP library (v.3.0.2) [41], a linear programming modeler.

Upon determining the optimal configuration, the *Node Selection Solver* transmits the solution to the Karpenter controller. Karpenter then provisions the *Spot Worker Node Pool* using the recommended

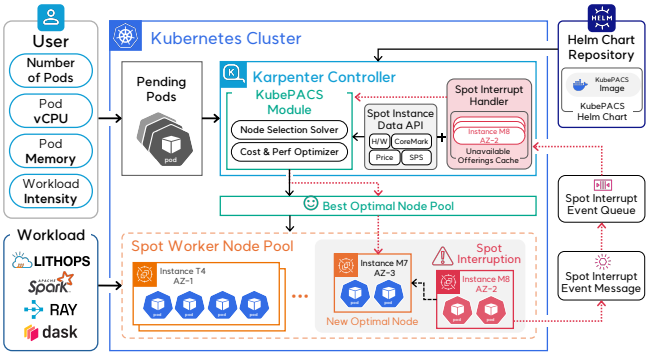


Figure 4: Implementation of KubePACS to provision Kubernetes worker nodes

instance types. New nodes subsequently join the cluster, allowing the pending pods to be rapidly scheduled.

4.1 Spot Interruption Handling Mechanism

A critical component of the KubePACS implementation is its robust mechanism for handling spot instance volatility. As depicted in Figure 4, spot interruption notifications from the cloud provider are captured as *Spot Interrupt Event Messages* and forwarded to a *Spot Interrupt Event Queue*. The *Spot Interrupt Handler* asynchronously processes these events and identifies the interrupted instance types. These interrupted offerings are immediately recorded in the *Unavailable Offerings Cache*. During the subsequent re-optimization cycle, the *Node Selection Solver* queries this cache to enforce constraints that exclude unstable offerings. Consequently, the system provisions a *New Optimal Node* that maximizes E_{total} while avoiding the interrupted availability zone or instance type. This reactive loop ensures rapid capacity recovery and maintains workload continuity.

4.2 Deployment and Distribution

To ensure ease of deployment and seamless integration into existing Kubernetes environments, KubePACS is packaged as a Docker container image and distributed via a standard *Helm Chart Repository* as the *KubePACS Helm Chart*. This packaging strategy allows administrators to deploy KubePACS alongside the Karpenter scheduler with minimal configuration overhead. To facilitate reproducibility and further research, the complete source code, Helm charts, and deployment artifacts are made publicly available.[1]

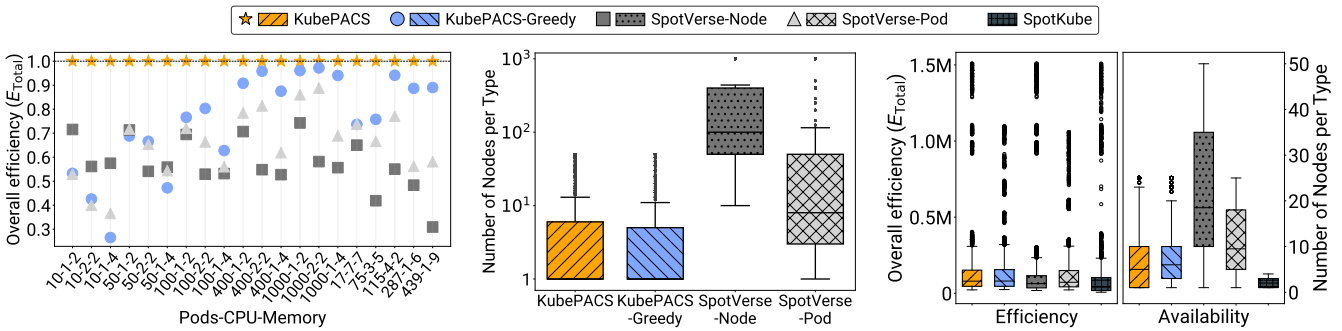
5 EVALUATION

The implemented framework is empirically evaluated to address the following research questions.

RQ-1 (Comparative Analysis): How does the KubePACS recommendation algorithm compare to state-of-the-art approaches (e.g., SpotVerse [55], SpotKube [15]) in terms of cost-efficiency, hardware performance, and availability?

RQ-2 (Design Validation): How do internal design choices of KubePACS, specifically the cost-performance hyperparameter (α) and workload-aware scaling, impact the system's effectiveness?

[1] https://kubepacs.ddps.cloud



(a) Overall efficiency (E_{Total}) **(b) Spot instance availability metric** **(c) Small-scale Microservice**

Figure 5: Comparing KubePACS with related works shows superb performance for cost and performance efficiency

RQ-3 (Real-world Impact): To what extent does KubePACS improve performance and reduce costs compared to production-grade Kubernetes baseline, Karpenter [12] with AWS SpotFleet?

5.1 Experiment Setup

Kubernetes Cluster Configuration Scenarios. To diversify the evaluation, we synthetically generate 15 Kubernetes cluster composition scenarios using the cartesian product of requested pod counts {10, 50, 100, 400, 1000} and pod configurations {(1 vCPU, 2 GiB), (2 vCPU, 2 GiB), (1 vCPU, 4 GiB)}. Each scenario is represented as a tuple `(Number of pods, vCPU, Memory)`. Additionally, we include 5 irregular configurations—{(17, 7, 7), (75, 3, 5), (115, 4, 2), (287, 1, 6), (439, 1, 9)}—resulting in a total of 20 scenarios.

Testbed Environment and Metrics. Datasets comprising spot and on-demand prices, benchmark scores, and SPS (both single- and multi-node) are acquired via SpotLake [38]. The collection period spans November 1–15, 2025, encompassing 731 instance types across all AZs in four AWS regions: N. Virginia, Oregon, Ireland, and Tokyo. For comparative analysis against a production-grade baseline, an Amazon EKS cluster is provisioned using Karpenter (v1.4) [12], with experiments conducted over a 12-hour window on May 22 in the corresponding regions. To simulate realistic application scenarios, Lithops [51] is employed to orchestrate compute-, network-, and disk-intensive tasks within the Kubernetes environment. The quantitative analysis focuses on the total hourly cluster cost and the efficiency metrics of $E_{PerfCost}$, $E_{OverPods}$, and E_{Total}.

5.2 Comparing KubePACS with State-of-the-Art

To address **RQ-1**, KubePACS is evaluated against the following baselines in terms of cost-efficiency, performance, and availability. *KubePACS-Greedy* serves as an ablation baseline, utilizing the identical dataset of KubePACS but employing a naive allocation strategy. Candidate instances are ranked by performance-cost efficiency ($E_{PerfCost}$), and pods are greedily allocated to top-ranked instances with the $T3$ constraint until the demand is met.

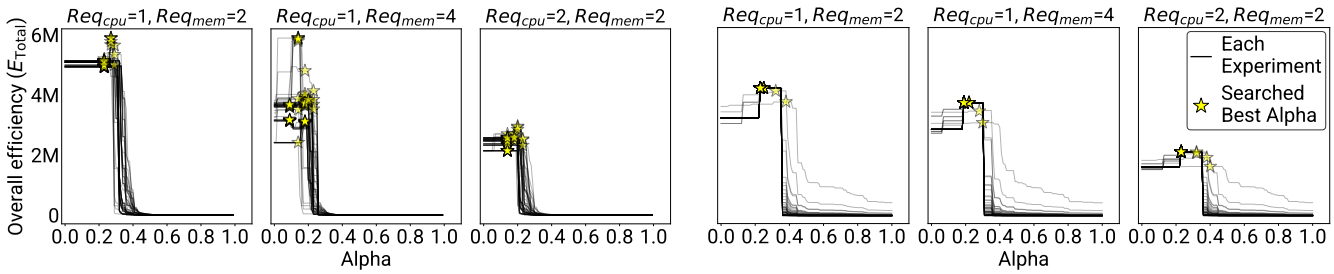
SpotVerse [55] filters candidates based on spot price, single-node SPS, and Interruption Frequency (IF). It filters instance–region pairs exceeding a combined SPS and IF scores over a threshold, prioritizing lower-priced options. Since SpotVerse originally operates at the instance level, two variants are adapted to align with Kubernetes pod semantics: *SpotVerse-Node* (lowest price per node) and *SpotVerse-Pod* (lowest price per pod).

SpotKube [15] employs an NSGA-II [19] genetic algorithm to optimize deployments. It utilizes a Pareto-based fitness function to balance cost against availability, enhancing resilience by distributing pods across diverse instance types and AZs.

Figure 5 presents a comparative analysis of KubePACS and related approaches. In Figure 5a, the y-axis represents the normalized overall efficiency (E_{Total}) across various pod requirement scenarios on the x-axis. The efficiency values are normalized relative to KubePACS, where a value of 1.0 serves as the reference; values below 1.0 indicate lower efficiency than KubePACS. Experimental results demonstrate that KubePACS outperforms all baselines, achieving average efficiency improvements of 48.11%, 81.06%, and 60.40% compared to *KubePACS-Greedy*, *SpotVerse-Node*, and *SpotVerse-Pod*, respectively. The reduced efficiency observed in *KubePACS-Greedy* is primarily attributed to excessive pod over-allocation, which negatively impacts $E_{OverPods}$. In contrast, the SpotVerse variants prioritize price and availability, failing to consider instance performance. Notably, *SpotVerse-Pod* exhibits substantial over-allocation, whereas *SpotVerse-Node* allocates fewer pods per node.

Figure 5b evaluates the availability implications of spot instance recommendations generated by each approach. To estimate the stability of spot instance requests, the number of allocated nodes per instance type is analyzed. This metric is motivated by the observation that excessive reliance on a single instance type significantly increases the risk of simultaneous interruptions, thereby reducing overall reliability [15]. The vertical axis represents the number of nodes allocated to each instance type, visualized using a box-and-whisker plot on a logarithmic scale. The experimental dataset comprises 4,800 samples collected over a 15-day period across four AWS regions, covering 20 distinct scenarios at six-hour intervals.

As illustrated in the figure, KubePACS effectively constrains the number of instances per type by utilizing the $T3$ metric as a strict upper bound. In contrast, SpotVerse does not impose such a constraint and frequently concentrates allocations onto a single instance type. This tendency is particularly pronounced in the *SpotVerse-Node* variant, which often recommends hundreds of identical instances. This lack of diversity exacerbates the risk of correlated failures, negatively impacting the aggregate availability of the cluster.



(a) AWS (b) Azure

Figure 6: Overall efficiency (E_{total}) changes with α, the cost-performance trade-off parameter

Comparison with SpotKube in Small-scale Scenarios. SpotKube was omitted from the extensive large-scale experiments as its original evaluation framework [15] is specifically designed for small-scale microservice environments. To ensure a fair baseline comparison, the experimental setup described in the SpotKube publication [15] was replicated. The workload involved pod counts ranging from 1 to 50, with each pod configured to require 1 vCPU and 1 GiB of memory. The candidate node pool was restricted to specific instance types of *t3.medium*, *c6a.large*, *t4g.large*, and *c6g.xlarge*.

Figure 5c presents the comparative results, with the left chart depicting overall efficiency (E_{Total}) and the right showing allocated instances per type. KubePACS achieves the highest efficiency, approximately 107% higher than SpotKube, primarily because SpotKube's rigid reliability mechanism enforces a fixed count of four instances per type, often forcing the selection of less efficient nodes to satisfy instance type diversity. In contrast, KubePACS employs a dynamic instance cap based on the $T3$ metric, effectively balancing cost-performance efficiency with robust spot availability guarantees.

Beyond KubePACS's superior efficiency, *KubePACS-Greedy* exhibits comparable performance, attributable to the limited cardinality of the candidate instance pool, which leads both algorithms to converge on similar recommendation sets. This highlights KubePACS's strength in large-scale public cloud environments, where its rigorous optimization becomes essential over greedy heuristics that suffice only for small and constrained search spaces.

5.3 Analysis of Internal KubePACS Mechanisms

This section analyzes the internal operational characteristics of KubePACS to address **RQ-2**.

Impact of Cost-Performance Weight Parameter (α). Figure 6 illustrates the variation of overall efficiency (E_{Total}) with respect to α during the GSS algorithm's exploration. The data were collected from 12 independent runs at 6-hour intervals between November 3–5, 2025, in AWS N. Virginia, and between February 16–18, 2026, in Azure US East. Black lines represent efficiency trajectories of explored node pools across varying α values, while yellow stars denote the α yielding maximum E_{Total} in each run.

As α increases from 0.0, which prioritizes spot cost exclusively, E_{Total} initially rises due to the selection of instances with superior hardware performance at marginal cost increase. However, exceeding the optimal α threshold causes a sharp decline in E_{Total}, resembling a step-down function, primarily driven by excessive pod over-allocation penalizing $E_{OverPods}$. Compared to the $\alpha = 0$ baseline mirroring cost-centric approaches, optimizing α improves E_{Total} by an average of 6% and up to 81%, demonstrating the advantage of incorporating hardware metrics into node recommendation.

Table 2: Normalized E_{Total} comparison across configurations

	Greedy	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1.0$	Ours
E_{Total}	0.8616	0.9563	0.0006	0.0001	**1.0000**

The same experiment on Azure, shown in Figure 6b, confirms cross-provider generalizability, as the characteristic concave pattern of E_{Total} with respect to α is consistently observed. However, Azure's absolute E_{Total} values are approximately 15% lower than AWS, attributable to limited SPS data coverage and lower number of available instance types: only 17.9% of candidate instance types maintained consistently valid SPS during the experimental period. Accordingly, the remaining experiments are conducted on AWS, where comprehensive spot instance data is available.

Table 2 quantifies the advantage of adaptive α optimization by comparing normalized E_{Total} across fixed α values and a greedy heuristic using the same input features as KubePACS. Fixed $\alpha = 0.5$ and $\alpha = 1.0$ suffer from severe over-provisioning, reducing E_{Total} to near zero, while the greedy approach achieves only 0.86 due to lack of global allocation control. This confirms that KubePACS's joint optimization, combining ILP-based allocation with adaptive α search via GSS, is essential for optimal cost-performance balance.

Selection of α Tolerance. Figure 7 analyzes the trade-off between optimal α search precision and ILP solver latency with respect to different tolerances, ε, based on 60 independent runs in the AWS N. Virginia region. As the error tolerance increases exponentially, the time to find the optimal α decreases linearly, which confirms the characteristics presented in Equation 6, but at the cost of reduced E_{Total}. We empirically found that a tolerance of 0.01 yields a good balance, reducing optimization time to approximately 2.0 seconds with negligible loss in recommendation quality.

System Overhead of the ILP Solver. The overhead of the ILP solver was profiled over 100 iterations per region. Peak memory consumption remains under 194 MB, and average CPU utilization is limited

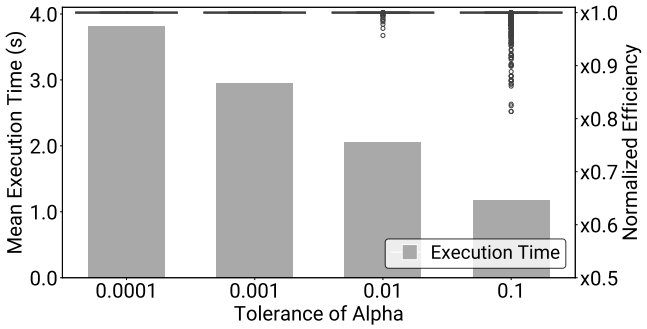


Figure 7: ILP solver latency and efficiency changes for different tolerance of α in the GSS algorithm

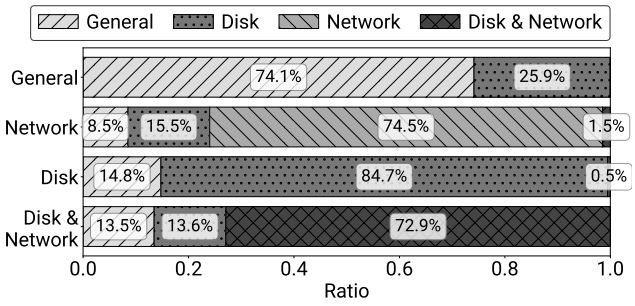


Figure 8: Effectiveness of special feature instance selection

to 1.55%, confirming that KubePACS introduces negligible overhead to the Kubernetes provisioning pipeline.

Effectiveness of Disk and Network Preferences. Figure 8 illustrates the distribution of instance types selected across different network or disk I/O preference settings. In the *General* scenario, where no specific preference is applied, general-purpose instances constitute the majority at 74.1%. However, disk-optimized instances still account for 25.9% of the selection; this is attributed to the system's cost-optimization logic, which opportunistically selects specialized instances when their spot prices drop below those of general-purpose instances. When a *Network* preference is specified, the system effectively selects network-optimized instances comprising 74.5% of the allocated nodes. Similarly, the *Disk* and *Disk & Network* scenarios demonstrate strong alignment with user intent, achieving 84.7% and 72.9% adherence to the respective specialized instance types. These results validate the efficiency of the proposed performance scaling mechanism, demonstrating its ability to accurately map user preferences to appropriate hardware configurations while maintaining cost efficiency.

Effectiveness of Multi-node SPS Metric. To validate the effectiveness of the multi-node SPS metric, experiments were conducted by requesting 50 instances across varying T3 values at hourly intervals over a 24-hour period (May 20–21, 2025). Figure 9 illustrates the correlation between T3 values and the number of successfully fulfilled nodes. The results exhibit a distinct positive trend, where higher T3 values correspond to significantly improved success rates

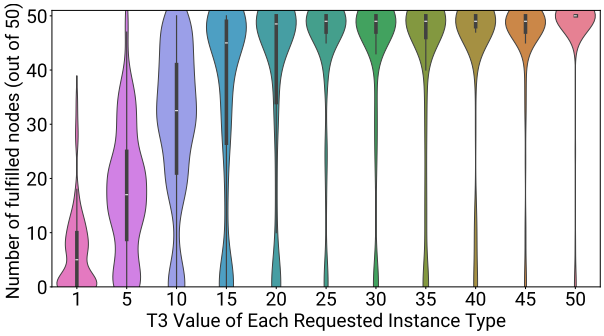


Figure 9: The number of fulfilled instances for different T3. The higher T3 ensures higher spot instance availability

in spot instance provisioning. These findings justify the integration of T3 as a reliability constraint within the ILP formulation. By enforcing these constraints, the proposed approach ensures enhanced stability for multi-node provisioning, offering fine-grained control over single-node-centric strategies, such as SpotVerse [55].

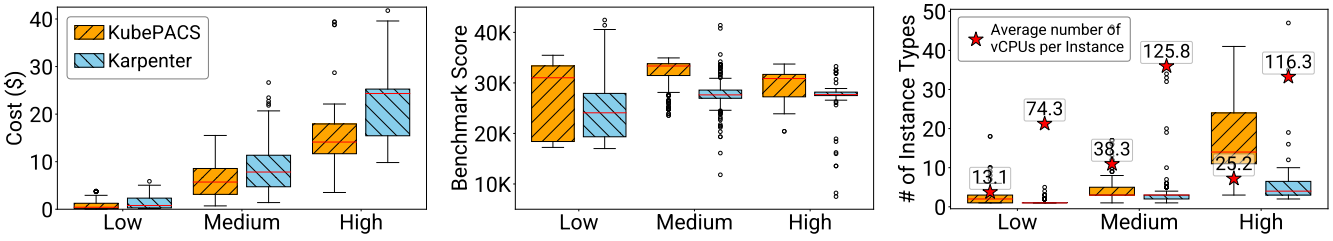
5.4 KubePACS in Real-World Environments

This section analyzes the performance of KubePACS within a realistic Kubernetes operation environment to address **RQ-3**.

5.4.1 Comparing KubePACS with Karpenter Scheduler. To validate the practical applicability of the proposed system, we conducted a comparative evaluation against Karpenter [12], a production-grade cloud instance provisioning system natively integrated with Kubernetes [50]. All experiments were conducted within the AWS using clusters subject to varying pod resource demands. Workloads were categorized into three distinct intensity levels based on their aggregate CPU and memory requirements: *Low* ($\leq$ 200 vCPUs, $\leq$ 200 GiB RAM), *Medium* ($\leq$ 800 vCPUs, $\leq$ 4000 GiB RAM), and *High* (exceeding the thresholds of the Medium category). Each scenario encompassed diverse pod configurations and was evaluated across four distinct AWS regions on August 19–20 and 22–23, 2025, as well as February 22, 2026, to capture temporal variations in spot market conditions. Each provisioning decision is independently optimized against the real-time market state at the moment of invocation, as KubePACS queries current T3 values and spot prices at each provisioning cycle. Figure 10 presents a comprehensive comparison of KubePACS and the Karpenter scheduler from the perspectives of cost-efficiency, instance performance, and spot instance availability.

Cost and Performance Efficiency. In terms of monetary cost, KubePACS consistently demonstrates superior cost-efficiency compared to Karpenter, as illustrated in Figure 10a. The experimental results indicate that KubePACS achieves an average cost reduction of 33% by effectively identifying cost-efficient spot instances.

The reduction in cost does not come at the expense of computational capability. As shown in Figure 10b, the instances recommended by KubePACS exhibit higher benchmark scores, surpassing Karpenter by an average of 12.15%. This dual advantage confirms that KubePACS successfully optimizes the trade-off between price and performance, selecting instances that are both cost-effective and high-performing.



(a) Total hourly cost (b) Recommended instances benchmark score (c) Spot instance availability-related metrics

Figure 10: Comparison of Cost, Performance, and Availability between KubePACS and Karpenter

Availability of Recommended Spot Instances. To evaluate the availability characteristics of recommended spot instances, we analyze two metrics: the cardinality of unique instance types (diversity) and the average vCPU core count per node (granularity). Prior work suggests that running workloads on a narrow set of instance types increases vulnerability to correlated interruptions [15], and larger instance types exhibit lower availability than smaller ones [38].

Figure 10c compares the diversity and granularity of instances recommended in each scenario. The vertical axis displays the distribution of unique instance types via box-and-whisker plots, while red star markers denote the average vCPU count. Karpenter tends toward consolidation, selecting few large-capacity instance types to satisfy resource demands, resulting in low type diversity and high average vCPU count. This increases interruption risk, as a single large node represents a substantial loss of computational resources [38], and reliance on a homogeneous instance set exacerbates availability risks when capacity shortages arise in a specific pool. In contrast, KubePACS mitigates these risks via *T*3-based constraints, distributing workloads across a diverse set of instance types. This diversification enables seamless substitution when specific pools experience capacity fluctuations, enhancing robustness and aggregate availability in spot-based environments.

5.4.2 KubePACS with Real-World Applications. To illustrate the practical benefits of KubePACS, we present three representative real-world use cases: (1) latency-sensitive compute-intensive services, (2) large-scale batch processing workloads, and (3) applications optimized for specialized network or disk I/O hardware. These scenarios demonstrate how KubePACS outperforms alternative approaches in terms of cost-efficiency, performance, and hardware-awareness.

Compute-Intensive Workloads. A prevalent use case for Kubernetes involves the deployment of long-running services, such as REST APIs [52]. These services typically adopts auto-scaling via HPA [6], which dynamically allocates additional pods based on runtime metrics (e.g., CPU utilization or incoming request rate).

For CPU-bound workloads, KubePACS facilitates the provisioning of high-performance instances at optimized costs while maintaining system reliability. To evaluate its efficiency, two representative compute-intensive REST services were deployed, a video encoding server based on ffmpeg [57] and a continuous integration build server utilizing the Rust development toolchain.

Table 3: Improvements for compute-intensive workloads

	Instance	Price/hour	App	Req./min	Price/Req.
Karpenter	c5.xlarge	0.0662$	Compilation	9	0.0074$
			Video enc.	31	0.0021$
KubePACS	c7i.xlarge	0.0733$	Compilation	13	0.0056$
			Video enc.	47	0.0016$
Best Case		**+10.73%**	Video enc.	**+51.61%**	**-23.8%**

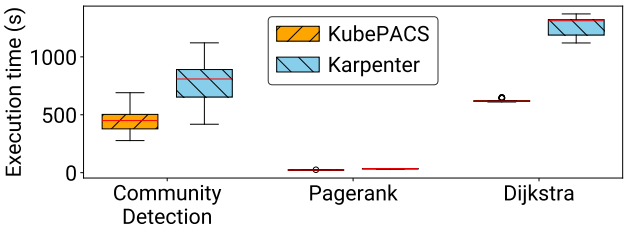
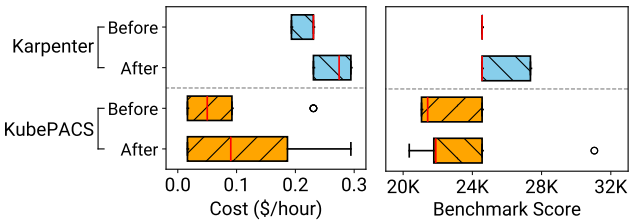


Figure 11: Execution times of graph analysis application

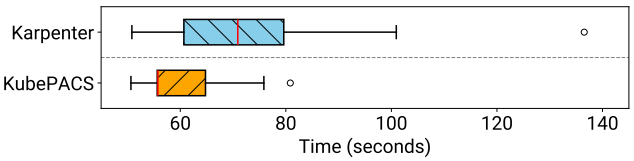
Table 3 presents a comparative analysis between KubePACS and Karpenter using a fixed pod configuration of 4 vCPUs & 8 GiB RAM. A one-pod-per-instance strategy was employed to strictly isolate performance effects at the instance level. The results indicate that KubePACS selects instances with significantly higher throughput, improving request processing rates by up to 51.61% (in the video encoding scenario), while incurring a marginal cost increase of 10.73%. Consequently, this yields an effective performance-per-dollar gain of up to 23.8% compared to Karpenter, while maintaining equivalent availability guarantees across both setups.

In scenarios where demand exceeds the capacity of existing pods, auto-scaling mechanisms dynamically provision additional resources. When configured to utilize KubePACS, the auto-scaler forms a fleet that inherits these aggregated benefits, ensuring a cluster that is more performant, cost-efficient, and highly available.

Batch Compute-Intensive Workloads. To assess KubePACS in batch-oriented environments, a graph analytics pipeline was implemented using the Lithops framework [51], processing large-scale graph datasets from object storage via compute-intensive algorithms including Community Detection, PageRank [44], and Dijkstra [14].



(a) Cost comparison **(b) Performance comparison**



(c) Recovery time comparison when interruption happens

Figure 12: The effectiveness of KubePACS interrupt handling

Figure 11 illustrates the distribution of execution times for each graph algorithm under distinct instance recommendation strategies. In the baseline configuration, Karpenter, adhering to the AWS SpotFleet price-capacity-optimized policy, allocated 12 *r4.xlarge* instances. Conversely, KubePACS identified and provisioned 12 *r6a.xlarge* instances, leveraging their superior computational throughput within comparable budget constraints. As a result, the cluster managed by KubePACS demonstrates significantly reduced execution latencies across all tested workloads. This performance advantage translates to a substantial improvement in cost-effectiveness, achieving gains of up to 51.55% compared to the Karpenter baseline.

I/O-Intensive Workloads. KubePACS effectively accommodates I/O-intensive applications by identifying and selecting instances optimized for network or disk performance, thereby enabling both higher throughput and cost efficiency in non-CPU-intensive scenarios. Network-intensive application performance is evaluated using an Extract-Transform-Load (ETL) pipeline that ingests data from Amazon S3, an object storage service. During a parallel download of 100 GB across 100 pods, KubePACS provisions n-type instances, achieving a 3.16× speedup compared to Karpenter, which relies on a generic provisioning policy lacking I/O awareness. This performance gain translates to an overall cost reduction of 52.44% for the data ingestion process.

For disk-intensive tasks, the system is evaluated using a file compression workload utilizing `tar` and `gzip`. KubePACS provisions d-type instances characterized by superior local disk I/O throughput, resulting in 4.92× faster execution relative to Karpenter, yielding a 79.86% reduction in cost. These findings validate the efficacy of the workload-aware instance selection mechanism employed by KubePACS across diverse I/O profiles.

While Karpenter allows users to manually specify preferred instance types with specialized capabilities (e.g., high-throughput networking or NVMe SSDs) to emulate the behavior of KubePACS, identifying and maintaining an exhaustive list of such types across a vast and evolving instance catalog is operationally prohibitive. In contrast, KubePACS enables users to declare workload intents at job submission time, automatically mapping these preferences to the optimal infrastructure. The development of autonomous workload profiling mechanisms that eliminate the need for explicit user inputs remains an important direction for future research.

5.4.3 Handling Spot Instance Interruptions. Figure 12 demonstrates the effectiveness of KubePACS's interrupt handling compared to Karpenter, where interruption events were manually injected using AWS *Fault Injection Service*. As shown in Figure 12a, KubePACS recommends significantly more cost-effective instances than Karpenter following an interruption event. Although Figure 12b indicates slightly lower hardware performance, this minor tradeoff is acceptable given the substantial cost savings. Furthermore, KubePACS recovers faster than Karpenter (Figure 12c), as Karpenter incurs considerable latency from calling SpotFleet service for recommendations, whereas KubePACS's solver overhead is negligible.

6 RELATED WORK

Compute cluster scheduling. Previous research has widely explored methods for cluster construction, scheduling, and workload organization, even without explicitly focusing on Kubernetes environments. Tetris [20] and Synergy [42] both assume pre-configured clusters and focus on efficiently allocating existing resources according to workload characteristics. In contrast, our work aims to construct workload-aware clusters from scratch in a cloud environment, achieving a balance between cost and performance. Stratus [11] and ExoSphere [53] select cost-effective spot instances primarily based on task resource demands, such as CPU and memory size, but do not consider other performance-related factors. Our study proposes a method for constructing a more efficient cluster by considering not only cost and resource requirements, but also characteristics such as per-core performance, network I/O, and disk I/O. Eva [8] dynamically optimizes the size and composition of a cloud-based cluster based on workload characteristics to achieve cost efficiency. It formulates the scheduling problem as an ILP, but solves it in practice using a reservation price-based heuristic that accounts for interference and migration overhead. However, it relies solely on on-demand instances, missing the additional cost savings that spot instances can provide.

Enhancing spot instance usage. Existing approaches typically optimize either cost or reliability. HotSpot [54] and Proteus [24] aim to reduce cost via dynamic migration or hybrid scheduling, but overlook performance variability. Tributary [23], Stratus [11], and Can't Be Late [60] enhance reliability through diversification or interruption modeling, yet disregard cost-performance trade-offs. SpotVerse [55] uses static thresholds on SPS and IF, which do not generalize to multi-node settings. SpotKube [15] jointly optimizes cost and reliability via a genetic algorithm, but lacks performance awareness and has only been tested at small scale. In contrast, this work proposes a multi-objective optimization framework that jointly considers cost, performance, and reliability. Unlike prior studies, it explicitly addresses the multi-node nature of cluster environments and utilizes real-world availability and benchmark data for instance selection, improving both scalability and practical applicability.

Table 4: Comparison of spot instance provisioning systems

	Karpenter	SpotKube	SpotVerse	Ours
Spot Price	✓	✓	✓	✓
SPS Awareness	–	–	Single-node	Multi-node
Benchmark Score	–	–	–	✓
Multi-obj. Optimization	–	Genetic Algo.	Threshold	ILP+GSS
Kubernetes Integration	✓	–	–	✓

Table 4 summarizes the key differences between KubePACS and existing spot instance provisioning systems. Data-wise, KubePACS uniquely incorporates multi-node SPS data and hardware benchmark scores alongside spot prices. Optimization-wise, while SpotKube employs a genetic algorithm and SpotVerse applies static threshold filtering, KubePACS formulates the problem as an ILP with adaptive hyperparameter tuning via GSS and workload-aware score scaling. Systems-wise, Karpenter provides native Kubernetes integration but lacks multi-objective optimization, whereas KubePACS embeds within the Karpenter controller to enable optimized spot instance provisioning with built-in spot interruption handling.

7 DISCUSSION AND FUTURE WORK

Several promising directions exist for extending the capabilities of KubePACS. While the current implementation relies on explicit user intents for instance selection, the integration of autonomous workload profiling mechanisms represents a key direction for future research. A substantial body of literature addresses workload performance profiling and prediction [25, 26, 37, 56]. Building on these foundations, we envision a two-phase approach: offline profiling that builds workload signatures from the resource utilization metrics of completed jobs, and online inference integrated into the Karpenter controller's provisioning loop to automatically classify incoming workloads. Incorporating such well-studied techniques would eliminate the need for manual specification by users, thereby enhancing usability and ensuring optimal resource mapping without human intervention.

The multi-objective optimization formulation can be extended to incorporate environmental sustainability metrics. Recent work has demonstrated carbon-aware datacenter operation at production scale [49], holistic carbon accounting across operational and embodied emissions [1], and lifecycle-wide environmental footprint measurement [22]. Building on these foundations, integrating carbon intensity data into the ILP constraints would allow KubePACS to balance cost, performance, and environmental impact while preferring lower-carbon regions and instance types, fostering green computing practices without changes to the optimization structure.

Since reducing the overall number of provisioned nodes directly lowers both operational energy consumption and embodied carbon, achieving such carbon reduction goals also motivates advancing the scheduler toward cluster-level joint optimization. Future iterations will address more complex scheduling scenarios, including the support for heterogeneous pod sizes and dynamic instance adjustment. Specifically, research will focus on vertical pod autoscaling mechanisms that adaptively resize allocations in response to fluctuating demand, as well as minimizing resource fragmentation when scheduling pods with diverse resource requirements on a shared node pool. By co-optimizing diverse workload types within a unified scheduler, overall node count can be reduced and resource efficiency improved, directly contributing to the carbon-aware scheduling objectives described above.

Furthermore, while the current implementation primarily targets AWS, extending KubePACS to support additional cloud providers such as Azure and GCP is a natural direction. The cross-provider experiments on Azure demonstrate that the ILP formulation generalizes effectively when analogous availability and pricing inputs are supplied, preserving the optimization behavior observed on AWS. Broadening this multi-cloud support would enable cross-provider orchestration, unlocking greater instance diversity and further opportunities for cost, performance, and carbon optimization.

Finally, extending KubePACS to support hardware accelerators, such as GPUs and TPUs [31], remains an important direction. Given the high volatility and cost of accelerator-based spot instances [10], adapting the availability scoring metric to account for accelerator-specific instance shortages and extending the scope of orchestration across multiple regions or clouds could yield substantial benefits for large-scale AI/ML workloads.

8 CONCLUSION

In this paper, we presented KubePACS, a Kubernetes-native spot instance provisioning engine designed to optimize the selection of instance types and the number of nodes by jointly considering cost-efficiency, hardware performance, and large-scale availability. To the authors' best knowledge, this is the first work to consider a wide range of aspects of cloud spot instances to guarantee high recommendation quality. By leveraging real-time multi-node-aware datasets including SPS, spot price, and benchmark metrics, KubePACS formulates the node selection process as a multi-objective optimization problem. The system efficiently searches for an optimal trade-off using a GSS algorithm and solves the instance recommendation problem via an ILP-based approach.

We implemented KubePACS as a Helm chart using a forked version of Karpenter and evaluated it through extensive experiments with both synthetic workloads and real-world applications. Compared to state-of-the-art methods, including SpotVerse [55], SpotKube [15], and AWS Karpenter [12], KubePACS consistently delivered higher performance per dollar, improved availability, and more efficient resource utilization. Our results demonstrate that intelligent, dataset-driven provisioning can unlock the full potential of spot instances in production Kubernetes clusters.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers and our shepherd, Rüdiger Kapitza, for their invaluable feedback. This work was supported by National Research Foundation (NRF) Grant funded by the Korea government (NRF-2020R1A2C1102544, RS-2023-00265538) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2022-00144309, RS-2025-25441560, RS-2026-25507506), the European Union through the Horizon Europe projects NEARDATA (101092644), CLOUDSTARS (101086248), and EXTRACT (101093110), and the Spanish Ministry of Science, Innovation and Universities through the X-AI project (PID2023-148202OB-C21).

REFERENCES

[1] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. 2023. Carbon Explorer: A Holistic Framework for Designing Carbon Aware Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 118–132. https://doi.org/10.1145/3575693.3575754

[2] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2013. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Trans. Econ. Comput.* 1, 3, Article 16 (sep 2013), 20 pages. https://doi.org/10.1145/2509413.2509416

[3] AWS. 2024. EC2 Fleet and Spot Fleet. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Fleets.html

[4] Azure. 2025. Spot Placement Score. https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/spot-placement-score

[5] Microsoft Azure. 2025. Compute benchmark scores for Azure Linux VMs. https://learn.microsoft.com/en-us/azure/virtual-machines/linux/compute-benchmark-scores.

[6] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2021. KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes. In *Service-Oriented Computing: 19th International Conference, ICSOC 2021, Virtual Event, November 22–25, 2021, Proceedings* (Dubai, United Arab Emirates). Springer-Verlag, Berlin, Heidelberg, 821–829. https://doi.org/10.1007/978-3-030-91431-8_59

[7] Matt Baughman, Simon Caton, Christian Haas, Ryan Chard, Rich Wolski, Ian Foster, and Kyle Chard. 2019. Deconstructing the 2017 Changes to AWS Spot Market Pricing. In *Proceedings of the 10th Workshop on Scientific Cloud Computing* (Phoenix, AZ, USA) *(ScienceCloud '19)*. Association for Computing Machinery, New York, NY, USA, 19–26. https://doi.org/10.1145/3322795.3331465

[8] Tzu-Tao Chang and Shivaram Venkataraman. 2025. Eva: Cost-Efficient Cloud-Based Cluster Scheduling. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1399–1416. https://doi.org/10.1145/3689031.3717483

[9] Yen-Ching Chang. 2009. N-Dimension Golden Section Search: Its Variants and Limitations. In *2009 2nd International Conference on Biomedical Engineering and Informatics*. 1–6. https://doi.org/10.1109/BMEI.2009.5304779

[10] Sungkyu Cheon, Kyumin Kim, Kyunghwan Kim, Moohyun Song, and Kyungyong Lee. 2025. Multi-Node Spot Instances Availability Score Collection System. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*. ACM.

[11] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. 2018. Stratus: cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 121–134. https://doi.org/10.1145/3267809.3267819

[12] Karpenter community. 2025. Karpenter : Just-in-time Nodes for Any Kubernetes Cluster. https://karpenter.sh/

[13] Standard Performance Evaluation Corporation. 2023. SPEC Benchmarks and Tools. https://www.spec.org/benchmarks.html.

[14] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[15] Dasith Edirisinghe, Kavinda Rajapakse, Pasindu Abeysinghe, and Sunimal Rathnayake. 2024. SpotKube: Cost-Optimal Microservices Deployment with Cluster Autoscaling and Spot Pricing. In *2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 87–94. https://doi.org/10.1109/CloudCom62794.2024.00026

[16] EEMBC. 2025. CPU Benchmark – MCU Benchmark – CoreMark – EEMBC Embedded Microprocessor Benchmark Consortium. https://www.eembc.org/coremark/.

[17] Nnamdi Ekwe-Ekwe and Adam Barker. 2018. Location, Location, Location: Exploring Amazon EC2 Spot Instance Pricing Across Geographical Regions. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 370–373. https://doi.org/10.1109/CCGRID.2018.00059

[18] Apache Software Foundation. 2004. Apache Hadoop. http://hadoop.apache.org/

[19] David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, New York.

[20] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/2619239.2626334

[21] Murli Gupta. 1991. Numerical Methods and Software (David Kahaner, Cleve Moler, and Stephen Nash). *Siam Review - SIAM REV* 33 (03 1991). https://doi.org/10.1137/1033033

[22] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S. Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2022. Chasing Carbon: The Elusive Environmental Footprint of Computing. *IEEE Micro* 42, 4 (July 2022), 37–47. https://doi.org/10.1109/MM.2022.3163226

[23] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1–14. https://www.usenix.org/conference/atc18/presentation/harlap

[24] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. 2017. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 589–604. https://doi.org/10.1145/3064176.3064182

[25] Darong Huang, Luis Costero, Ali Pahlevan, Marina Zapater, and David Atienza. 2024. CloudProphet: A Machine Learning-Based Performance Prediction for Public Clouds. *IEEE Transactions on Sustainable Computing* 9, 4 (2024), 661–676. https://doi.org/10.1109/TSUSC.2024.3359325

[26] Yoonseo Hur and Kyungyong Lee. 2024. CNN Training Latency Prediction Using Hardware Metrics on Cloud GPUs. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 216–226. https://doi.org/10.1109/CCGrid59990.2024.00033

[27] David Irwin, Prashant Shenoy, Pradeep Ambati, Prateek Sharma, Supreeth Shastri, and Ahmed Ali-Eldin. 2019. The Price Is (Not) Right: Reflections on Pricing for Transient Cloud Servers. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*. 1–9. https://doi.org/10.1109/ICCCN.2019.8846933

[28] AWS What is New. 2021. Introducing Amazon EC2 Spot placement score. https://aws.amazon.com/about-aws/whats-new/2021/10/amazon-ec2-spot-placement-score/

[29] Bahman Javadi, Ruppa K. Thulasiram, and Rajkumar Buyya. 2013. Characterizing spot price dynamics in public cloud environments. *Future Generation Computer Systems* 29, 4 (2013), 988–999. https://doi.org/10.1016/j.future.2012.06.012 Special Section: Utility and Cloud Computing.

[30] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[32] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 83–91. https://doi.org/10.1145/3038912.3052707

[33] KyungHwan Kim and Kyungyong Lee. 2024. Making Cloud Spot Instance Interruption Events Visible. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 2998–3009. https://doi.org/10.1145/3589334.3645548

[34] Kyunghwan Kim, Subin Park, Jaeil Hwang, Hyeonyoung Lee, Seokhyeon Kang, and Kyungyong Lee. 2023. Public Spot Instance Dataset Archive Service. In *Companion Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 69–72. https://doi.org/10.1145/3543873.3587314

[35] Primate Labs. 2025. Geekbench 6 - Cross-Platform Benchmark. https://www.geekbench.com/.

[36] C. C. Lee and D. T. Lee. 1985. A Simple On-Line Bin-Packing Algorithm. *J. ACM* 32, 3 (jul 1985), 562–572. https://doi.org/10.1145/3828.3833

[37] Sungjae Lee, Yoonseo Hur, Subin Park, and Kyungyong Lee. 2022. PROFET: PROFiling-based CNN Training Latency ProphET for GPU Cloud Instances. In *2022 IEEE International Conference on Big Data (Big Data)*. 186–193. https://doi.org/10.1109/BigData55660.2022.10020212

[38] S. Lee, J. Hwang, and K. Lee. 2022. SpotLake: Diverse Spot Instance Dataset Archive Service. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 242–255. https://doi.org/10.1109/IISWC55918.2022.00029

[39] Aniruddha Marathe, Rachel Harris, David Lowenthal, Bronis R. de Supinski, Barry Rountree, and Martin Schulz. 2014. Exploiting Redundancy for Cost-Effective, Time-Constrained Execution of HPC Applications on Amazon EC2. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/2600212.2600226

[40] Robert McGill, John W Tukey, and Wayne A Larsen. 1978. Variations of box plots. *The American Statistician* 32, 1 (1978), 12–16.

[41] Stuart A. Mitchell. 2003. PuLP. https://github.com/coin-or/pulp.

[42] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 579–596. https:

//www.usenix.org/conference/osdi22/presentation/mohan

[43] Danielle Movsowitz Davidow, Orna Agmon Ben-Yehuda, and Orr Dunkelman. 2023. Deconstructing Alibaba Cloud's Preemptible Instance Pricing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 253–265. https://dl.acm.org/doi/pdf/10.1145/3588195.3593001

[44] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[45] S. Gopal Krishna Patro and Kishore Kumar Sahu. 2015. Normalization: A Preprocessing Stage. *CoRR* abs/1503.06462 (2015). arXiv:1503.06462 http://arxiv.org/abs/1503.06462

[46] Google Cloud Platform. 2025. CoreMark scores of VM instances by family. https://cloud.google.com/compute/docs/coremark-scores-of-vm-instances.

[47] Gustavo Portella, Genaina N. Rodrigues, Eduardo Nakano, and Alba C.M.A. Melo. 2019. Statistical analysis of Amazon EC2 cloud pricing models. *Concurrency and Computation: Practice and Experience* 31, 18 (2019), e4451. https://doi.org/10.1002/cpe.4451 e4451 cpe.4451.

[48] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing. Chapter 10.1. Golden Section Search in One Dimension* (3 ed.). Cambridge University Press, USA.

[49] Ana Radovanović, Ross Koningstein, Ian Schneider, Bokan Chen, Alexandre Duarte, Binz Roy, Diyue Xiao, Maya Haridasan, Patrick Hung, Nick Care, et al. 2022. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems* 38, 2 (2022), 1270–1280.

[50] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale.* 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[51] Josep Sampé, Marc Sánchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro García-López. 2023. Outsourcing Data Processing Jobs With Lithops. *IEEE Transactions on Cloud Computing* 11, 1 (2023), 1026–1037. https://doi.org/10.1109/TCC.2021.3129000

[52] Shazibul Islam Shamim, Jonathan Alexander Gibson, Patrick Morrison, and Akond Rahman. 2022. Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes. arXiv:2211.07032 [cs.SE] https://arxiv.org/abs/2211.07032

[53] Prateek Sharma, David Irwin, and Prashant Shenoy. 2017. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1, 1 (2017), 1–23.

[54] Supreeth Shastri and David Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 493–505. https://doi.org/10.1145/3127479.3132017

[55] Myungjun Son, Gulsum Gudukbay Akbulut, and Mahmut Taylan Kandemir. 2024. SpotVerse: Optimizing Bioinformatics Workflows with Multi-Region Spot Instances in Galaxy and Beyond. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 74–87. https://doi.org/10.1145/3652892.3700750

[56] M. Son and K. Lee. 2018. Distributed Matrix Multiplication Performance Estimator for Machine Learning Jobs in Cloud Computing. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Vol. 00. 638–645. https://doi.org/10.1109/CLOUD.2018.00088

[57] Suramya Tomar. 2006. Converting video formats with FFmpeg. *Linux J.* 2006, 146 (June 2006), 10.

[58] Cheng Wang, Qianlin Liang, and Bhuvan Urgaonkar. 2017. An Empirical Analysis of Amazon EC2 Spot Instance Features Affecting Cost-Effective Resource Procurement. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3030207.3030210

[59] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. 2017. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/3126908.3126953

[60] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. 2024. Can't Be Late: Optimizing Spot Instance Savings under Deadlines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 185–203. https://www.usenix.org/conference/nsdi24/presentation/wu-zhanghao