Future Generation Computer Systems 176 (2026) 108230

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Predicting runtime and resource utilization of jobs on integrated cloud and HPC systems

Esma Yildirim [a,*], Mohab Hussein [a], Mikhail Titov [b], Ozgur Ozan Kilic [b]

[a] Mathematics and Computer Science, Queensborough Community College of CUNY, 222-05 56th Ave, Bayside, 11364, NY, USA
[b] Computational Science Initiative, Brookhaven National Laboratory, PO Box 5000, Upton, 11973, NY, USA

ARTICLE INFO

Keywords:
Batch job scheduling
Neural networks
Performance modeling
Measurement and prediction of multi-processor systems
Cloud and HPC resource monitoring

ABSTRACT

Recent advances in virtualization technologies used in cloud computing offer performance that closely approaches bare-metal levels. Combined with specialized instance types and high-speed networking services for cluster computing, cloud platforms have become a compelling option for high-performance computing (HPC). However, most current batch job schedulers in HPC systems are designed for homogeneous clusters and make decisions based on limited information about jobs and system status. Scientists typically submit computational jobs to these schedulers with a requested runtime that is often over- or under-estimated. More accurate runtime predictions can help schedulers make better decisions and reduce job turnaround times. They can also support decisions about migrating jobs to the cloud to avoid long queue wait times in HPC systems.

In this study, we design neural network models to predict the runtime and resource utilization of jobs on integrated cloud and HPC systems. We developed two monitoring strategies to collect job and system resource utilization data using a workload management system and a cloud monitoring service. We evaluated our models on two Department of Energy (DOE) HPC systems and Amazon Web Services (AWS). Our results show that we can predict the runtime of a job with 31–41% mean absolute percentage error (MAPE), 14–17 seconds mean absolute value error (MAE), and 0.99 R-squared (R^2) score. Having an MAE of less than a minute corresponds to 100% accuracy since the requested time for batch jobs is always specified in hours and/or minutes.

1. Introduction

Current batch schedulers in HPC systems typically make job scheduling decisions based on user-requested runtimes and basic system load information (e.g., number of available nodes). However, user-provided runtime estimates are often inaccurate, and limited information is available regarding a job's I/O behavior at submission time. When users overestimate runtimes, schedulers may allocate resources inefficiently, making suboptimal decisions compared to scenarios with accurate runtime data. Underestimations can be even more problematic, leading to wasted system resources and user time. In contrast, a scheduler capable of accurately predicting a job's actual runtime and I/O load can make significantly more optimized decisions. One recent study employs convolutional neural networks (CNNs) to analyze job script data and the system's current disk I/O load to predict actual job runtimes [1]. The goal is to enable intelligent co-scheduling while avoiding disk I/O contention. However, the proposed model is limited to homogeneous HPC systems and only considers disk I/O metrics for load estimation.

Another study [2] introduces a reinforcement learning-based scheduler that adapts to environmental changes, such as shifts in workload characteristics. The model leverages data from scheduler logs–including submit time, wait time, requested runtime, and number of compute nodes or CPU cores. However, the only system load metric considered is the number of available nodes, and I/O contention is not addressed. This model is also designed solely for homogeneous systems. Similarly, the study in [3] applies reinforcement learning to support scheduling decisions for both immediate and reserved job executions, incorporating backfilling to improve resource utilization. It estimates node availability to reserve resources for long-running, high-priority jobs, yet remains limited to homogeneous system configurations.

One of the key gaps in the current literature is that models leveraging Artificial Intelligence (AI) techniques–such as reinforcement learning and neural networks–for intelligent scheduling or runtime prediction are designed exclusively for homogeneous systems. A homogeneous system consists of identical processing cores and a uniform architecture. In contrast, a heterogeneous system incorporates multiple types of

* Corresponding author.
E-mail addresses: eyildirim@qcc.cuny.edu (E. Yildirim), mohab.hussein95@qmail.cuny.edu (M. Hussein), mtitov@bnl.gov (M. Titov), okilic@bnl.gov (O.O. Kilic).

https://doi.org/10.1016/j.future.2025.108230
Received 15 April 2025; Received in revised form 23 July 2025; Accepted 26 October 2025
Available online 3 November 2025

processing cores. An integrated system extends this further by combining multiple distinct systems, which may differ in interconnects, storage architectures, and hardware components, but operate under a unified or compatible software environment. Furthermore, HPC systems operate under numerous allocation constraints–for example, not all resources are available for every allocation type. Accurate job runtime prediction is critical due to the presence of multiple job queues (e.g., small, medium, large), and misclassification can lead to premature job termination. These limitations are typically absent in cloud environments, where users can configure systems to meet specific requirements, and once deployed, operate without such restrictions.

On the other hand, HPC systems often feature cutting-edge processors, high-speed networks, and advanced storage architectures, which can offer significant advantages, particularly for highly parallel or I/O-intensive workloads. Moreover, the software ecosystems–such as job schedulers and file systems–differ significantly between HPC and cloud environments, with HPC platforms generally being more mature in this regard. Nevertheless, if we invest in developing intelligent workload management systems tailored for cloud platforms, there is little to prevent HPC centers from transitioning to predominantly cloud-based infrastructures.

Models that account for heterogeneous or integrated distributed systems (e.g., HPC and cloud platforms) can support the development of more adaptable schedulers and facilitate seamless job migration from HPC systems to cloud environments, thereby improving overall efficiency. Moreover, collecting a broader set of system-level metrics during workload execution can significantly enhance the accuracy of predicted values such as job runtime or system I/O load. In a recent study [4], a multivariate time series model based on neural networks was employed to predict data transfer throughput in cloud networks, achieving an error rate of just 3.7 %. The model utilized multiple system metrics–including CPU utilization, network I/O rate, and disk I/O rate–gathered from various cloud components. This approach outperformed traditional univariate models, demonstrating the value of using multivariate system utilization data to improve the predictive accuracy of cloud resource performance models.

Artificial neural networks (ANNs) have a well-established theoretical justification for modeling complex non-linear relationships: Hornik et al. [5] showed that even shallow multilayer feedforward networks are universal function approximators, capable of capturing arbitrarily complex mappings across inputs and outputs. As time passed, several algorithms were proposed to train these networks better [6]. In this study, we designed an artificial neural network model to predict the runtime and resource utilization of a job or a bag of tasks on integrated cloud and HPC systems. The model takes as input the runtime of the job/task on either system, along with load information from both systems. This enables accurate decision-making regarding job migration–specifically, whether moving a job from a congested HPC system to a cloud platform can reduce waiting time–and helps estimate the cost of executing the job on the alternative system, particularly when using a commercial cloud provider.

To create representative workloads, we used a combination of benchmark suites and real-world applications modeled after those typically executed on HPC systems in national laboratories, where the latest supercomputing technologies are deployed. The workloads were generated using a randomized program we developed, and executed on both HPC and cloud platforms under varying parameters (e.g., number of CPUs/GPUs, threads, nodes, and application-specific settings). During execution, we collected metrics related to jobs/tasks and system utilization. The neural network model was trained using this dataset to predict runtime and resource utilization .

Our model achieved runtime prediction on the cloud system with a MAPE of 31–41 %, a MAE of 14–17 seconds (i.e., under one minute), and an R^2 score of 0.99. Additionally, we predicted CPU utilization on the cloud system with 2.4-2.7 % MAE and an R^2 score of 0.95, and GPU utilization on the HPC system with 0.075-0.089 % MAE and an R^2 score of 0.99. The contributions of this work include:

- A workload generator architecture capable of creating randomized workloads with diverse characteristics, submitting them to target systems, and monitoring and measuring system resource utilization of jobs and tasks.
- A unique time series dataset capturing resource utilization and system load metrics for jobs and tasks executed across multiple HPC and cloud platforms.
- A neural network model that accurately predicts the runtime and resource utilization of a job on a second system, given its runtime on the first system and the load information from both environments.

The rest of the paper is organized as follows: Section 2 describes the architecture of the workload generator and the data collection methods; Section 3 introduces the prediction model; Section 4 outlines the experimental methodologies and presents the results; Section 5 compares related work; and Section 6 concludes the paper.

2. Workload generator architecture

2.1. Workload design

A workload refers to a collection (or bag) of related or unrelated jobs/tasks, each with specific application characteristics. In HPC systems, administrators often adjust the configuration or behavior of the batch job scheduler in response to shifts in the types and patterns of jobs submitted over time. The first goal of this study was to develop a method for generating workloads that reflect the day-to-day usage patterns observed in DOE HPC systems, which typically feature hybrid architectures incorporating both CPUs and GPUs. Prior studies have generally taken one of two approaches: either simulating workload execution [1], or using historical workload logs in experimental evaluations [2]. The simulation-based approach offers flexibility in configuring system parameters, but it struggles to realistically capture the behavior and impact of real-world workloads on system load. Conversely, log-based approaches provide data grounded in actual usage, but the available metrics are limited and often insufficient to comprehensively model system and job-level dynamics.

When designing a model for use across multiple systems with differing architectures and characteristics, it is essential to replicate tests under controlled conditions on each system. In our study, we executed the designed workloads–comprising both benchmarks and real-world applications–on HPC and cloud platforms using varied parameter settings (e.g., parallelism level, job arrival time, job type such as long-running vs. short-running, and problem size). This variability in configuration introduces diversity into the input data fed to the neural network model, which improves the model's ability to generalize and increases prediction accuracy. To achieve this, we incorporated a selection of benchmarks from the NAS Parallel Benchmarks suite [7], as well as machine learning applications such as image classification tasks [8], which target GPU-based architectures using TensorFlow [9].

2.1.1. Benchmarks

The selected benchmarks include suites that scale efficiently using the MPI [10] and OpenMP [11] programming paradigms, and utilize a range of HPC system resources, such as CPU, memory, and disk I/O. We chose a subset of the NAS Parallel Benchmarks (NPB) suite based on the architectural and performance requirements of the HPC and cloud systems used in this study.

The NAS Parallel Benchmarks are a set of compact problems derived from computational fluid dynamics (CFD) applications and are designed to evaluate the performance of parallel supercomputers. The original suite consists of five computational kernels and pseudo-applications, which are available in both MPI and/or OpenMP implementations:

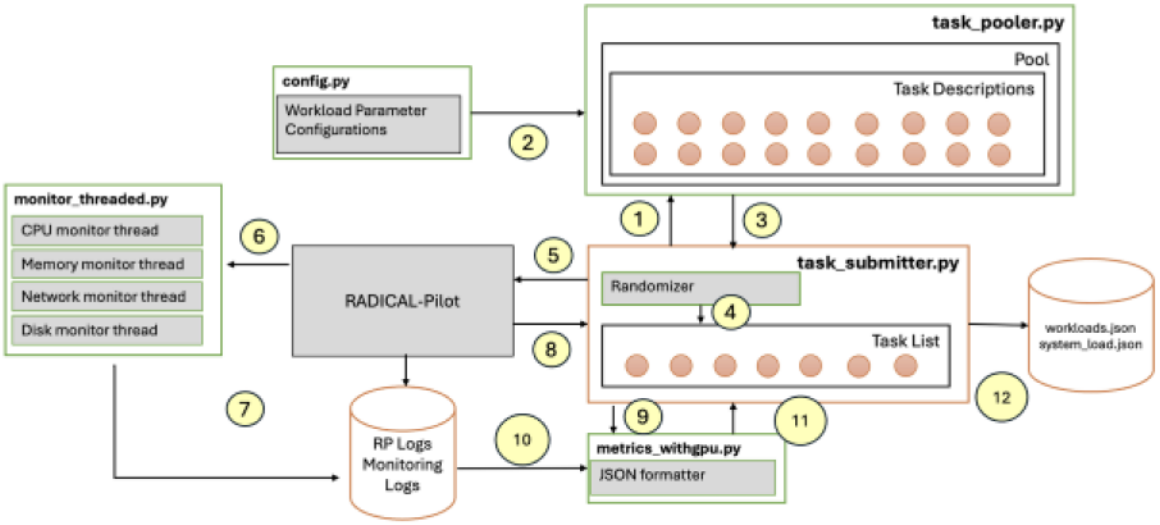

Fig. 1. Workload generator architecture using RP.

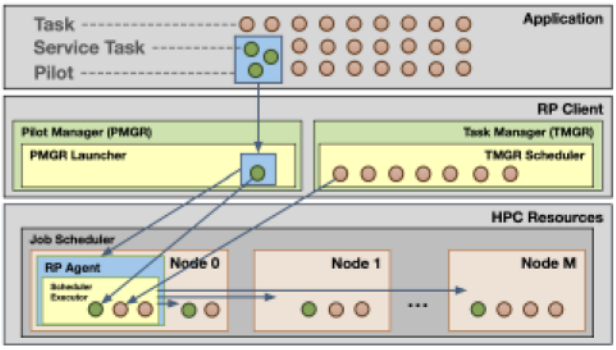

Fig. 2. RP components and management processes for the workload execution.

Five kernels:

- IS - Integer Sort, random memory access, OpenMP, MPI
- EP - Embarrassingly Parallel, OpenMP, MPI
- CG - Conjugate Gradient, irregular memory access and communication, OpenMP, MPI
- MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive, OpenMP, MPI
- FT - discrete 3D fast Fourier Transform, all-to-all communication, OpenMP, MPI

Three pseudo applications:

- BT - Block Tri-diagonal solver, OpenMP, MPI
- SP - Scalar Penta-diagonal solver, OpenMP, MPI
- LU - Lower-Upper Gauss-Seidel solver, OpenMP, MPI

Benchmarks for unstructured computation, parallel I/O, and data movement:

- UA - Unstructured Adaptive mesh, dynamic and irregular memory access, OpenMP

We also included the 3 benchmarks from the multizone suites (NPB-MZ) to test hybrid MPI + OpenMP cases:

- BT-MZ - Block Tri-diagonal solver, uneven-size zones within a problem class, increased number of zones as problem class grows
- SP-MZ - Scalar Penta-diagonal solver, even-size zones within a problem class, increased number of zones as problem class grows
- LU-MZ - Lower-Upper Gauss-Seidel solver, even-size zones within a problem class, a fixed number of zones for all problem classes

These benchmarks are available in multiple problem sizes, referred to as class sizes. For our experiments, we primarily selected Class C, and in some cases Class D, to achieve moderate runtime durations on both HPC and cloud systems. Additionally, some applications impose specific constraints on the number of parallel tasks, such as requiring a perfect square or a power of two, which were taken into account during workload configuration.

2.1.2. Image classification with tensorflow/keras

As deep learning (DL) workloads increasingly appear in HPC environments, their unique characteristics introduce new challenges for system performance and resource management [12]. To reflect this emerging trend, we selected an image classification task designed to run on both CPUs and GPUs [8], and modified it to support experimentation with various parameters, such as batch size, dataset size, and number of training epochs. The application utilizes the 786 MB Kaggle Cats vs. Dogs dataset [13] and implements a binary classification model based on the Xception architecture [14]. To explore the impact of input size on performance and resource utilization, we partitioned the dataset into three categories: small, medium, and large. The original dataset serves as the large category, while the other two were generated by halving the number of image files sequentially. The batch size was treated as a tunable parameter to accommodate varying memory capacities across systems. Similarly, increasing the number of epochs extended the runtime but improved model accuracy. To leverage thread-level parallelism and utilize multiple GPUs per node, we incorporated TensorFlow's *MirroredStrategy*. However, we excluded the use of *MultiWorkerMirroredStrategy* due to its incompatibility with CPU-only architectures operating under the Slurm job scheduler [15].

2.2. Resource monitoring and metric collection

We believe that the most effective way to learn the characteristics of a workload–such as its runtime on a given system–is by analyzing its system resource usage (e.g., CPU utilization, bytes transmitted and received across all network interfaces, etc.).

Before designing our own metric collection system, we examined existing workload archives in the literature [16–18] and found that none of them adequately addressed the problem we aim to solve. The Parallel Workload Archive [16], for example, only provides information on requested resources such as the number of CPUs and memory size,

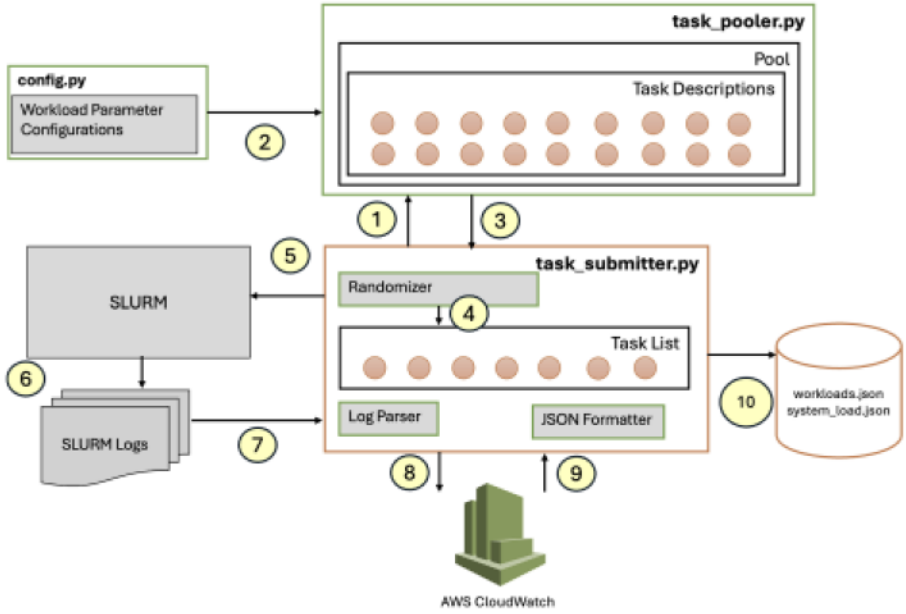

Fig. 3. Workload generator architecture using CloudWatch.


```
[
  {
    "workload-name": "small-tasks",
    "platform": "cloud_aws",
    "tasklist": [
      {
        "task_id": 0,
        "session_id": "rp.session.ip-10-0-0-171.ec2-user.019909.0001",
        "app_name": "image_classification_wGPU_monitor.py_bs64_dssmall_ep1",
        "submit_time": 1720190071.0035641,
        "cpus": 31,
        "gpus": 0,
        "start_time": "1720190062.3628280",
        "finish_time": "1720190387.8327400",
        "nodes": [
          {
            "node_name": "queue1-st-c6a8xlarge-2",
            "metrics": {
              "cpu_util": [
                [
                  "1720190062.4142168",
                  "11.6"
                ],
                ...
              ],
              "network_util": [
                [
                  "1720190062.4221125",
                  "1445888",
                  "732247"
                ],
                ...]]
```


Fig. 4. Workload dataset format.

while actual job-level resource usage is missing from most datasets. Additionally, the archive does not allow for the identification of the same job executed under different configurations, as each job instance is associated with a unique ID. However, job identification across systems and settings is essential for training neural network models to predict runtime and resource utilization on multiple platforms. A similar limitation is present in the Google Cluster Data archive [17], where job tracking across different platforms and settings is not possible. We also considered the Grid Workloads Archive [18], but the dataset is no longer available. Given these limitations, we developed a new metric collection system capable of producing time series resource usage data, while enabling consistent job identification across HPC and cloud systems.

Cloud and HPC systems offer different tools and capabilities for metric collection, resulting in varying sets of available metrics on each platform. To address this, we identified comparable workload and system load metrics and developed automated methods to collect these metrics during workload execution using tools and services such as RADICAL-Pilot [19] and AWS CloudWatch [20].

RADICAL-Pilot (RP) is a pilot-based system written in Python and specialized in executing applications composed of many heterogeneous computational tasks on HPC platforms. As a Pilot system, RP separates resource acquisition from using those resources to execute application tasks. Resources are acquired by submitting a job to the batch system of an HPC machine. Once the job is launched on the requested resources, RP can directly schedule and launch application tasks on those resources. Thus, tasks are not scheduled via the batch system of the HPC platform, but directly on the acquired resources with the maximum degree of concurrency they afford. CloudWatch is the monitoring service provided by AWS and it can provide different monitoring metrics for the services used in the cloud with a default 5-minute interval.

To mimic the day-to-day workloads of DOE Leadership Computing Facilities (LCFs), we designed a randomized workload generator using RP and AWS CloudWatch, implemented on two different system architectures. Fig. 1 illustrates the steps of the first architecture for workload generation and metric collection, annotated with yellow circled numbers. At the core of this design are the Task Submitter and RP. In Step 1, the Task Submitter invokes the Task Pooler to generate a pool of task descriptions. These are based on configuration parameters tailored to different workload types. For instance, to generate a workload with large tasks, the configuration includes Class D NAS benchmarks and image classification tasks with large epoch counts or large dataset sizes. In Step 2, the Task Pooler uses this configuration file to generate task descriptions by enumerating all possible parameter combinations and returns the task pool to the Task Submitter (Step 3). In Steps 4 and 5, the Task Submitter randomly selects a user-defined number of tasks and submits them to RP. Simultaneously, it defines an RP service task for monitoring system resources, including CPU, memory, network, and disk usage. Because of job scheduler restrictions, GPU monitoring is added separately as a thread within the GPU-using application. This is necessary because GPUs can only be monitored if they are explicitly allocated to a task, and service tasks are not permitted to consume significant system resources. RP then executes both the application and

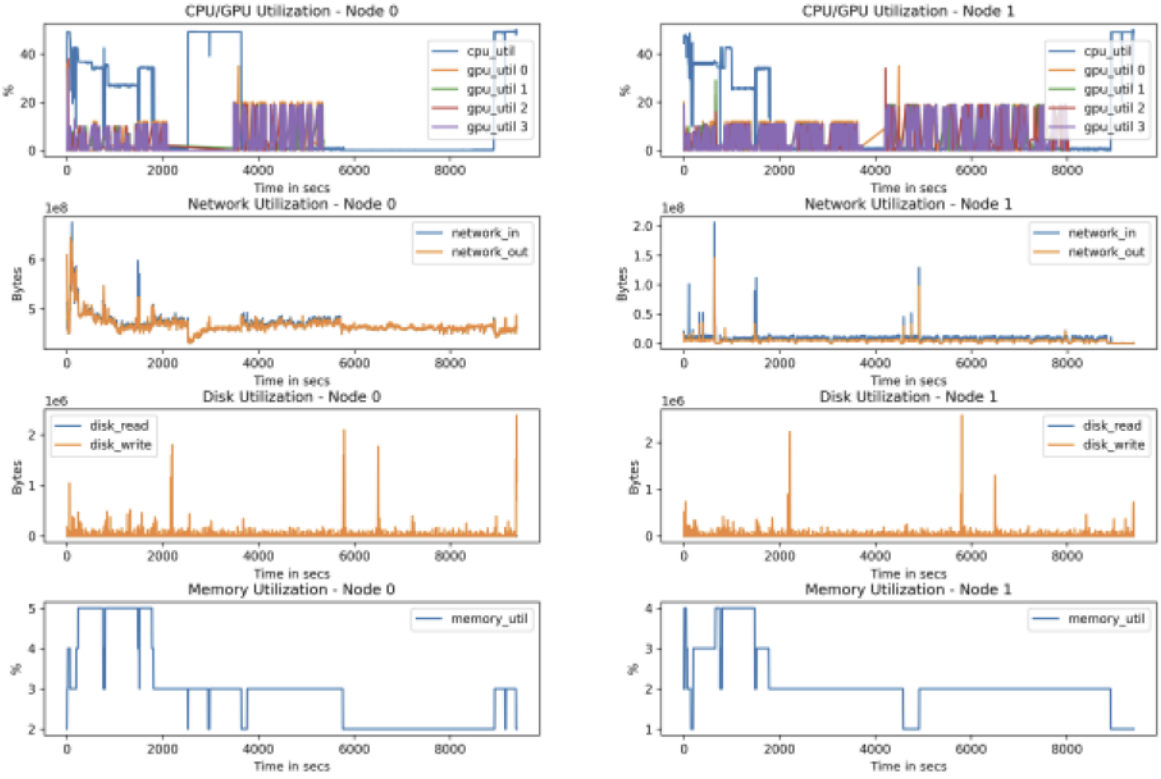

Fig. 5. Sample mixed workload system resource utilization on IC2 cluster.

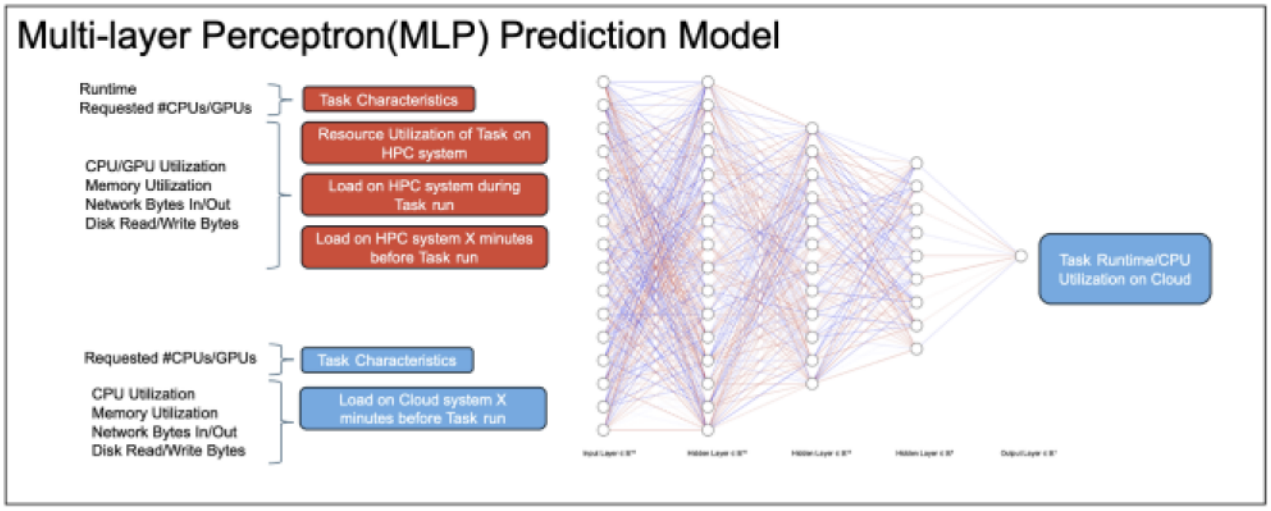

Fig. 6. Prediction model.

service tasks, while storing logs in separate files (Steps 6 and 7). Once the application tasks complete, the service tasks are terminated, and control is returned to the Task Submitter (Step 8). The Task Submitter then uses *metrics_withgpu.py* to process the monitoring logs (Steps 9, 10, and 11) and writes the results into two JavaScript Object Notation (JSON) files: one for workload-level metrics and one for system load data (Step 12). The JSON file format is described in Section 2.3.

The metrics collected from the HPC and cloud systems using this architecture include CPU and GPU utilization, memory utilization, number of bytes transmitted and received across all network interfaces, and number of bytes read from or written to disk. To support this, we developed a multi-threaded monitoring application using a combination of available Nvidia, Linux, and Python tools and libraries. After evaluating several options, we selected the following tools for data collection: *netstat*, *iostat*, *collectl*, *free*, *nvidia-smi*, and Python's *psutil* library. All metrics are recorded as time series data to enable fine-grained temporal analysis of resource utilization.

Measuring CPU Utilization: Although there are other tools available that can report CPU utilization as a percentage (e.g., *vmstat*, *top*), we preferred using Python libraries whenever possible, as the workload generator is implemented in Python. This approach also minimizes discrepancies that may arise due to variations in tool outputs across different systems. The *psutil* library was chosen to measure CPU utilization at fixed intervals. Specifically, we collect utilization data from the past 1 second, then pause for 4 seconds to complete a 5-second monitoring interval. Each data point is logged with an attached timestamp.

Measuring GPU Utilization: On cluster nodes equipped with GPUs, the *nvidia-smi* tool can be used to access GPU information; however, it only

```
config = { "nas":
           {   "paradigm"  : ["mpi", "openmp", "hybrid"],
               "class"     : ["C", "D"],
               "openmp"    : {
                   "max_threads" : 32
                   },
               "mpi"       : {
                   "nodes" : 4
                   },
               "hybrid"    : {
                   "nodes" : 4
                   },
               "iterations": 3
            },

           "image_classification":
           {
                "epochs"    : [1, 5, 10],
                "data_size" : ["small", "medium", "large"],
                "batch_size": [32, 64],
                "gpus_per_rank": [1, 2, 4]
                "iterations": 25
           }
}
```

Fig. 7. Mixed workload configuration.

provides output if the GPUs are allocated. Allocating GPUs solely for monitoring purposes would prevent them from being used by the application tasks. To address this limitation, we implemented a monitoring thread within each GPU-using application task. This thread parses the output of *nvidia-smi* and writes it to the RP task logs, accompanied by timestamps.

Measuring Memory Utilization: This metric, like CPU and GPU utilization, is measured as a percentage and indicates the proportion of memory bytes currently in use. The memory utilization monitoring thread invokes the *free* tool, parses its output, and logs the results along with a timestamp. The thread then sleeps for 5 seconds before repeating the process.

Measuring Network In/Out: These two metrics describe the number of bytes received and transmitted on all network interfaces of a node. Measuring this metric proved to be one of the most challenging tasks, as different systems offered different toolsets. Initially, we used *ifstat*, which was available on most systems. However, during our tests, we observed that *ifstat* produced implausibly large values, particularly when CPU utilization reached 100%. As a result, we replaced it with *netstat* and calculated the bytes received/transmitted by multiplying the number of packets by the maximum transmission unit (MTU) size. Unfortunately, *netstat* only reports cumulative values. To obtain measurements in 5-second intervals, we invoked the tool both before and after the sleep interval and computed the difference between the two readings. On systems where *netstat* was unavailable, we employed an alternative tool, *collectl*.

Measuring Disk Read/Write: These two metrics describe the number of bytes read from and written to disk by I/O calls to the operating system. The *iostat* tool is available on both Cloud and HPC systems and offers options to report average values over the past *n* seconds and for a specified number of readings. We observed that when using these options, the first reading always remained constant, with changes appearing only in subsequent outputs. To address this, we configured the tool to perform two readings and discarded the first. The parsed metrics were then written to the log file along with a timestamp.

We created a service task description for the monitoring application (*monitor_threaded.py*) and attached it to the RP pilot description of the workload application. RP allocated a single core to the service task on each node (the green task in Fig. 2) and launched it across all nodes in the allocation. Since the service task is bound to a CPU core by default, the monitoring application's threads run on a single core, thereby avoiding unnecessary system resource consumption. The remaining resources are reserved for regular workload tasks. While the tasks execute, the monitoring threads measure system resource utilization and write the data to log files. Task-level metrics such as start time, finish time, and the list of assigned nodes are collected from RP's client and agent logs, whereas system utilization metrics are extracted from the per-node log files written by the monitoring service tasks. The code for the workload generator is available in our GitHub repository [21], with system-specific configuration variations. The workload generator can also be extended to support applications beyond the NAS benchmarks and the Image Classification workload by supplying the appropriate task descriptions and configuration parameters.

Fig. 3 shows the second architecture, where we utilize the underlying batch job scheduler (SLURM [15]) to schedule tasks and AWS CloudWatch to monitor resource utilization. This architecture is specific to the AWS Cloud, whereas the first architecture is applicable to both cloud and HPC systems. Similar to the first architecture, the Task Submitter invokes the Task Pooler to generate a comprehensive pool of task descriptions based on configuration parameters tailored to various workload types. However, in Step 5, instead of submitting tasks to RP, the Task Submitter submits them to the SLURM scheduler for execution. Once all jobs in the queue have completed and the logs are written (Step 6), the process moves to Step 7, where SLURM logs are retrieved. For this architecture, a primary focus is placed on extracting relevant job status information from the *slurmctld.log* file generated by the SLURM scheduler on an AWS ParallelCluster. This log file contains key details about job scheduling and execution, including job start and completion times, job IDs, node allocations, and other status metrics. The metric collection process begins by parsing the *slurmctld.log* file. By extracting these details, we can accurately determine the duration and performance characteristics of each job.

Using the parsed start and stop times of jobs in Step 7, the Task Submitter invokes a series of functions through the *boto3* library to query the CloudWatch service for specific performance metrics during the job execution period (Steps 8 & 9). The primary metrics collected include network, CPU, and storage-related statistics:

- *CPU utilization*: Time series data representing the average CPU utilization (in percentage) over a specific interval (e.g., 1-minute or 5-minute intervals, as provided by CloudWatch). This metric is collected from all compute node instances.
- *Network In/Out*: Time series data showing the average number of bytes received and sent by each compute node instance.
- *Volume Write/Read Bytes*: Time series data representing the average number of bytes written to or read from the Elastic Block Storage (EBS) volumes attached to the compute nodes.

Unfortunately, memory utilization data is only available for the head node in CloudWatch. As a result, we were unable to collect memory metrics for the compute nodes.

Overall, this systematic approach to metric collection enables a comprehensive analysis of job performance, offering insights into network activity, CPU utilization, and storage I/O operations. Finally, in Step 10, the collected data is formatted into two JSON files, following the same structure as in the first architecture.

2.3. Dataset format

The metrics collected from cloud and HPC systems are unified using the JSON data format for ease of use and analysis. We define two data formats: (1) *workloads* and (2) *system load*. The *workloads* format is designed to represent workload characteristics such as task submit time, start time, finish time, number of CPUs/GPUs requested, and the resource utilization of the nodes on which each task is executed. Fig. 4 shows an excerpt from a *workloads.json* file collected from an AWS ParallelCluster composed of *c6a.8xlarge* instances, each equipped with 32 vCPUs. The utilization data is parsed from the system node logs generated by our workload generator architecture during the time window

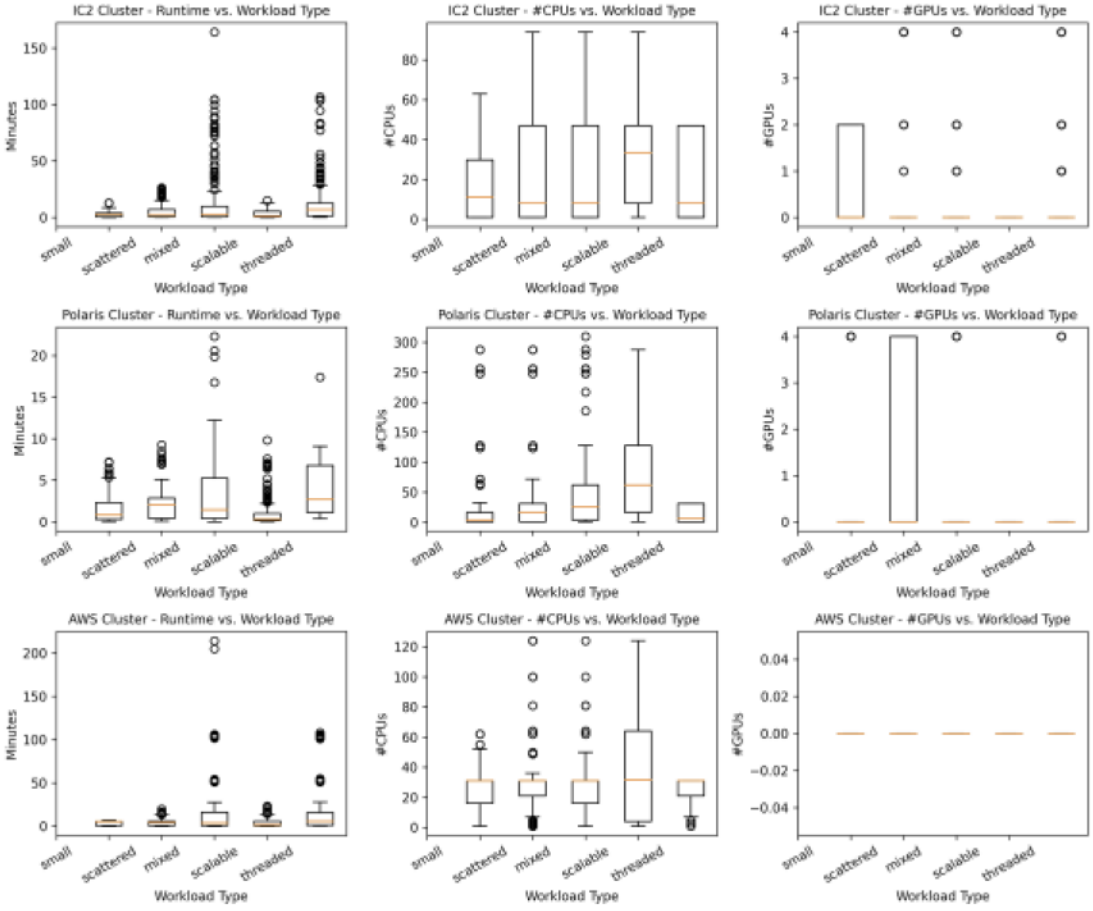

Fig. 8. Workload statistics.

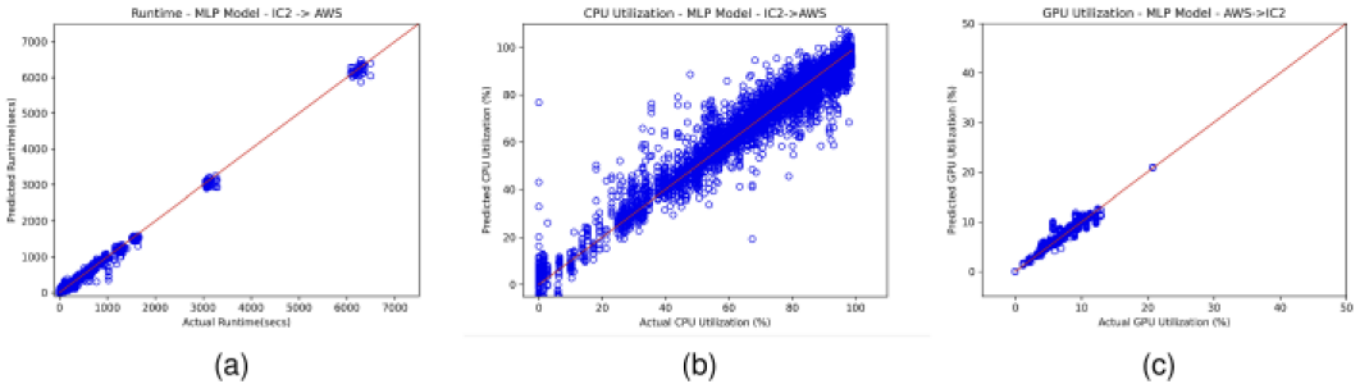

Fig. 9. Prediction results using RP monitoring logs - system 1: IC2 - System 2: AWS PC 1.

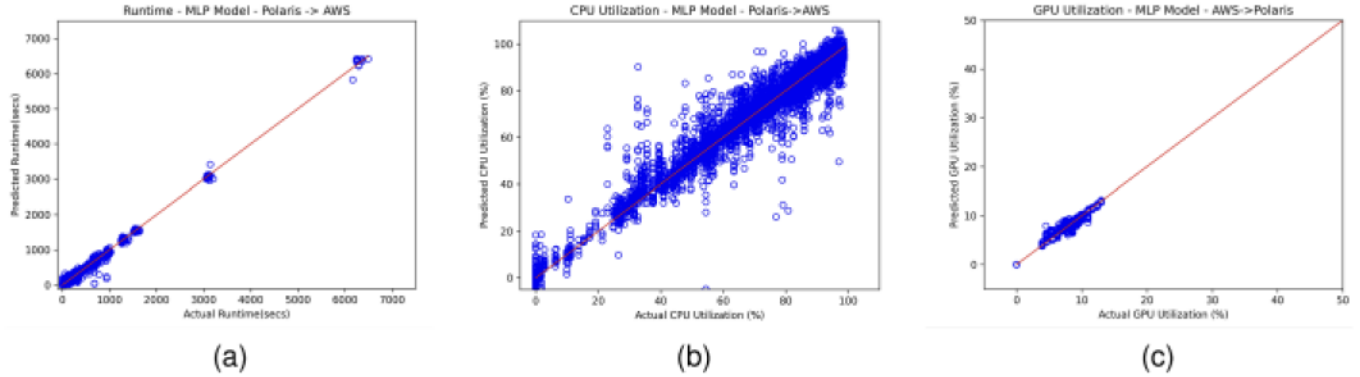

Fig. 10. Prediction results using RP monitoring logs - system 1: polaris - system 2: AWS PC 1.

between the task's start and finish times. These utilization values are averaged over the nodes assigned to the task. In the CPU utilization list, each item is a pair containing the timestamp of the measurement and the corresponding CPU utilization value. Memory utilization is recorded in the same format. As shown in the figure, the network utilization list includes entries with a timestamp, followed by network-in and network-out values in bytes. Disk utilization is similarly represented; however, instead of network in/out, the entries contain disk read and write measurements in bytes.

The second data format is designed to represent the total system load of the nodes allocated by RP at the beginning of the execution, regardless of where individual tasks are running. The motivation for separating the *workloads* and *system load* formats is to facilitate the creation of an AI-ready dataset, enabling straightforward integration with machine learning models. Fig. 5 illustrates system resource utilization from a sample mixed workload, as captured in the *system_load.json* file from the IC2 cluster at Brookhaven National Laboratory (BNL). The time series format allows us to monitor changes in resource utilization over time and observe correlations among various metrics. For instance, in the first half of the graphs, we observe that CPU utilization increases in parallel with memory and network utilization on both nodes. The trends in network input and output bytes generally align, though the spikes in network in are more pronounced. Additionally, disk write activity is visible, while disk read values remain mostly zero. The versatility of this time series data structure makes the dataset applicable to a wide range of problems beyond the scope of this study–for example, forecasting resource utilization for job scheduling and management. Therefore, we believe this dataset will be a valuable resource for the scientific community. The dataset is publicly available on the Zenodo platform (DOI: https://doi.org/10.5281/zenodo.15545095).

3. Prediction model

We have tested several neural networks architecture types including Multi-level Perceptrons (MLPs), CNNs, Recurrent Neural Networks (RNNs), and Long Short-term Memory Networks (LSTMs), however our best results were obtained with an MLP model, therefore, we only explain the details of the MLP model in this section. We have identified 33 input features that might be related to the output parameter we would like to predict (e.g., runtime, CPU utilization, etc.). Table 1 shows the list of the input features selected.

Fig. 6 illustrates the input and output features used in the MLP model. The red features are collected from the HPC system, while the blue ones are obtained from the AWS cloud. We designed the model as a regression model consisting of three hidden layers, using *he_uniform* for weight initialization, the Rectified Linear Unit (ReLU) as the activation function in the hidden layers, and a linear activation function in the output layer. The model was trained with the Adam optimizer for 100 epochs, using Mean Squared Error (MSE) as the loss function. It was configured to predict either the runtime or the CPU/GPU utilization of the corresponding job. To address the potential curse of dimensionality, we performed a correlation analysis using Pearson's Rho, Spearman's Rho, and Kendall's Tau, aiming to exclude highly correlated input features. However, the best results were achieved when using all input features. Models that only included correlated variables showed reduced performance. The code for the model is available in our GitHub repository [21].

4. Experimental results

In this section, we present the characteristics of the workloads generated and executed on various HPC and Cloud systems, along with the accuracy of our prediction model for estimating the runtime and CPU/GPU utilization of jobs and tasks.

Table 1
Model features.

System 1	
1	Runtime of job/task
2	#CPUs requested
3	#GPUs requested
4	CPU utilization of job/task
5	GPU utilization of job/task
6	Memory utilization of job/task
7	Network In (Mbps)
8	Network Out (Mbps)
9	Disk Read (Mbps)
10	Disk Write (Mbps)
11	Total CPU utilization of system X minutes before job/task run
12	Total GPU utilization of system X minutes before job/task run
13	Total Memory utilization of system X minutes before job/task run
14	Total Network In (Mbps) of system X minutes before job/task run
15	Total Network Out (Mbps) of system X minutes before job/task run
16	Total Disk Read (Mbps) of system X minutes before job/task run
17	Total Disk Write (Mbps) of system X minutes before job/task run
18	Total CPU utilization of system during job/task run
19	Total GPU utilization of system during job/task run
20	Total Memory utilization of system during job/task run
21	Total Network In (Mbps) of system during job/task run
22	Total Network Out (Mbps) of system during job/task run
23	Total Disk Read (Mbps) of system during job/task run
24	Total Disk Write (Mbps) of system during job/task run
System 2	
25	#CPUs requested
26	#GPUs requested
27	Total CPU utilization of system X minutes before job/task run
28	Total GPU utilization of system X minutes before job/task run
29	Total Memory utilization of system X minutes before job/task run
30	Total Network In (Mbps) of system X minutes before job/task run
31	Total Network Out (Mbps) of system X minutes before job/task run
32	Total Disk Read (Mbps) of system X minutes before job/task run
33	Total Disk Write (Mbps) of system X minutes before job/task run

4.1. Systems

We used two HPC systems–IC2 cluster from BNL and Polaris from Argonne National Laboratory (ANL)–along with two cloud clusters created using the AWS ParallelCluster (PC) tool. Table 2 shows the number of nodes allocated and the number of CPUs/GPUs available per node on each system. While IC2 and Polaris feature heterogeneous node types with both CPUs and GPUs, we selected CPU-only clusters on AWS using *c6a.8xlarge* and *t2.8xlarge* instances to evaluate the performance differences when migrating an application that can run on both GPU and CPU to a CPU-only environment. Both IC2 and AWS ParallelCluster systems use the SLURM scheduler on the backend, while the Polaris cluster uses the PBS scheduler [22], leading to configuration differences in the RP installation across systems.

4.2. Workloads

To mimic shifts in workload characteristics, we created workloads in five different categories: *small*, *scattered*, *mixed*, *scalable*, and *threaded*. A *small* workload consists of applications with their smallest problem

sizes–corresponding to the C class in the NAS Parallel Benchmark and a small dataset with 1 epoch in the image classification application. In the *scattered* category, we aimed for applications with runtimes closer to one another, using C and D class benchmarks from NAS and small datasets with 1 to 5 epochs for image classification. The *mixed* category includes all parameter combinations. The *scalable* category focuses on applications that can scale across multiple nodes, such as MPI and hybrid MPI + OpenMP benchmarks from the NAS suite. The *threaded* category includes only OpenMP benchmarks from NAS and the image classification application with all possible parameter settings.

In addition to application-specific parameters, we varied the number of CPUs, GPUs, and nodes requested to represent different levels of parallelism. A sample configuration for the *mixed* category is shown in Fig. 7. The thread and node numbers represent the maximum values, and all possible values less than or equal to these maximums are generated in the task pool. We employed an exponential increment strategy for generating these values. Since the valid parameter sets differ by application, an *iterations* parameter is used to specify what percentage of task descriptions from each application are included in the task pool.

Each workload consists of 50 applications (a.k.a. tasks) and is generated using five different random seeds across five different categories by the randomizer in the Task Submitter, in order to represent varied arrival orders and parameter combinations. A total of 1250 applications/tasks were executed on each system, generating input datasets with approximately ≈ 40,000 data points for our prediction model. It is very difficult to model the high variability in real-world workload arrival times; therefore, researchers often rely on representative statistical values such as mean, variance, and others [23,24]. According to [2], when jobs arrive sparsely, their waiting time in the queue becomes zero, making it difficult for scheduling policies–especially those based on machine learning models–to learn meaningful patterns. In such cases, any policy may appear effective regardless of its quality. Taking this observation into account, we designed a busy arrival schedule in which tasks enter the system following a Poisson distribution with a mean inter-arrival time of 10 seconds. This choice reflects the presence of many short-running benchmarks in our workload.

Fig. 8 presents the mean and standard deviation of runtime, number of CPUs, and number of GPUs for each workload type across all systems. Given the number of nodes allocated in each cluster, it is expected that workloads on the Polaris system exhibit the lowest runtimes due to the scalability offered by its 10-node configuration, compared to 2 and 4 nodes on the IC2 and AWS clusters, respectively. Interestingly, the IC2 cluster demonstrates lower runtimes than the AWS cluster despite having fewer nodes. We attribute this to the availability of GPUs on the IC2 cluster, which accelerates execution.

Among the workload types, the *small* workloads exhibit the shortest mean runtimes, following the *scalable* workloads, owing to the efficient parallelization of scalable applications across available resources. In contrast, the *threaded* workloads have the highest mean runtimes, as their parallelism is restricted to a single node on all systems. The *mixed* workloads display mean runtimes that fall between the other categories, indicating that our parameter configuration achieved a balanced distribution across workload types.

When we examine the number of CPUs in the second column, we observe that the mean values for all workload types are centered around the maximum number of CPUs available on the AWS cluster. On the IC2 cluster, the mean CPU values are generally lower compared to the other clusters, except for the *scalable* workloads. This is likely due to the use of GPUs in all other workload types on IC2, whereas the *scalable* workloads utilize only CPUs. On Polaris, the mean CPU values are higher, which can be attributed to the larger number of allocated nodes.

GPU statistics are easier to interpret since the applications use only thread-level parallelism. The mean GPU values are zero across all clusters, primarily because NAS parallel benchmark applications comprise 75 % of the workload, compared to 25 % for the image classification application. This imbalance is intentional, as the image classification application has significantly higher runtimes than the NAS benchmarks. On the IC2 cluster, the *small* workload type exhibits notable GPU usage, while the *scattered* workload on Polaris shows a similar pattern. However, due to limited memory on Polaris, we had to use the maximum number of GPUs per node (4) for feasible execution. This constraint explains why GPU outliers on Polaris consistently show 4 GPUs, whereas on the IC2 cluster, a broader mix of 1, 2, and 4 GPUs was achievable. The AWS cluster shows no GPU data points, as it lacks GPU-enabled instances. These observations confirm that we achieved a diverse and balanced configuration of applications with varying runtimes and CPU/GPU resource requirements.

Table 2
HPC and cloud systems.

Name	#nodes alloc.	#CPUs/node	#GPUs/node
IC2-SDCC-BNL	2	48	4
Polaris-ALCF-ANL	10/25/50	32	4
AWS PC 1	4	32	0
AWS PC 2	4	8	0

Table 3
Total runtime ratios ($Runtime \times \#CPUs/GPUs$).

Application Type	IC2 / AWS PC 1	Polaris / AWS PC 1	Polaris / IC2
Scalable (MPI)	1.00	3.06	3.23
Threaded (OpenMP)	1.45	0.95	0.63
Hybrid (MPI + OpenMP)	1.21	0.89	0.75
Thread (TensorflowGPU)	0.09	0.04	0.47

An additional noteworthy observation emerged when we compared the total runtimes of jobs, calculated as the product of runtime and the number of CPUs/GPUs used, across different systems for each job type. Table 3 presents the runtime ratios between system pairs across various job categories. Although the IC2 and AWS clusters offer 96 and 128 CPU cores respectively, IC2 performed comparably to AWS for scalable jobs and underperformed for threaded and hybrid jobs. As expected, GPU-dependent jobs ran faster on IC2, given that the AWS cluster was CPU-only. The Polaris system showed slightly better performance than AWS for OpenMP and hybrid jobs but performed significantly worse for scalable workloads. We suspect this is due to increased communication overhead at higher parallelism levels on Polaris, compared to AWS. For GPU-dependent jobs, Polaris's performance was similar to that of IC2. When comparing Polaris and IC2 directly, Polaris outperformed IC2 across most job categories, except for scalable jobs. These findings suggest a potential advantage in moving toward cloud-based systems. However, more extensive testing on larger-scale cloud clusters is necessary before drawing any definitive conclusions.

4.3. Prediction results

In this section, we present the evaluation results of the MLP model used to predict the runtime, CPU utilization, and GPU utilization of jobs/tasks when migrated from one system to another. Since the neural network is designed as a regression model, we employ standard evaluation metrics including MAE, MSE, MAPE, and R^2 score. MAE calculates the average absolute difference between actual and predicted values, as shown in Eq. 1, where N is the number of data points, y is the actual output, and $\hat{y}$ is the predicted output. MAE is the most robust metric in the presence of outliers. In contrast, MSE uses the square of the differences between actual and predicted values (Eq. 2), which causes larger errors to be penalized more heavily. This makes MSE more sensitive to outliers compared to MAE. MAPE measures the average absolute percentage error between predictions and actual values (Eq. 3) and is also highly sensitive to outliers, especially when actual values are close to

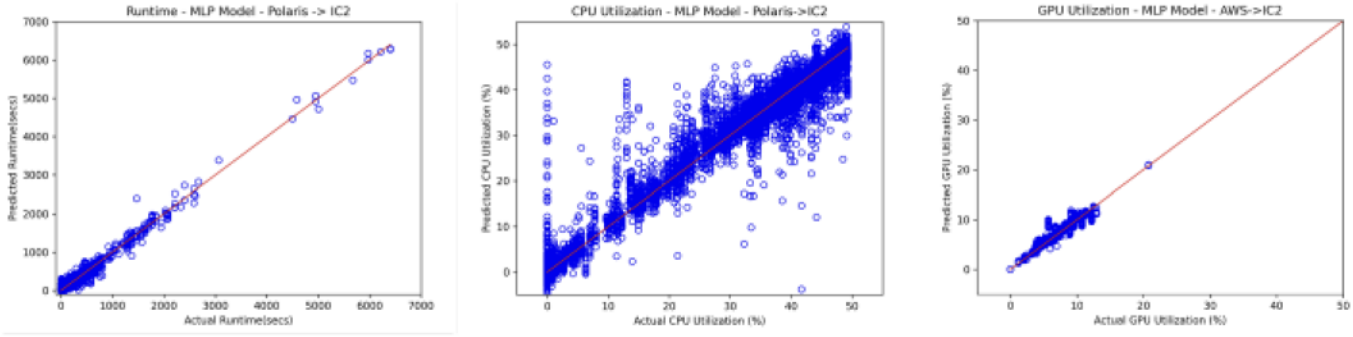

Fig. 11. Prediction results using RP monitoring logs - system 1: polaris - system 2: IC2.

zero. For MAE, MSE, and MAPE, lower values indicate better model performance.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y-\hat{y}| \tag{1}$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y-\hat{y})^2 \tag{2}$$

$$MAPE = \frac{\sum_{i=1}^{N}\frac{|y-\hat{y}|}{y}\times 100}{N} \tag{3}$$

$$R^2 = 1-\frac{SSR}{SST} \tag{4}$$

Unlike the previously discussed error metrics, R^2–also known as the coefficient of determination–is used to assess the goodness of fit of a regression model. It measures the proportion of the variance in the dependent variable that can be explained by the independent variables in the model. A higher R^2 value indicates that the model better captures the underlying data patterns and provides stronger explanatory power. R^2 is calculated as shown in Eq. 4, where SSR denotes the sum of squared residuals between the predicted and actual values, and SST represents the total sum of squares, which reflects the total variance present in the actual output values. The closer the R^2 value is to 1, the more effectively the model explains the variability in the dataset.

We used these four evaluation metrics, along with scatter plots of predicted vs. actual values, to assess how accurately the model predicts outcomes. Fig. 9 presents the scatter plots for jobs migrated to the AWS cluster, based on data collected from both the IC2 and AWS clusters using the RP monitoring architecture. In Fig. 9.a, prediction points for runtime are concentrated around the 45-degree line, indicating high prediction accuracy. However, there are visible gaps between clusters of data points, suggesting that runtime values are not uniformly distributed. We attribute this to the exponential parameter scaling strategy used when configuring workload variations. For CPU and GPU utilization predictions, the data points are similarly aligned along a linear trend line (Figs. 9.b and 9.c, respectively). We observed that GPU utilization typically remains below 20 %, while CPU utilization can reach up to 100 %. The model achieved a Mean Absolute Error (MAE) of 17 seconds, a Mean Absolute Percentage Error (MAPE) of 31 %, and an R^2 score of 0.99 for runtime prediction. Considering that the average runtime for AWS workloads is approximately 8 minutes, a 17-second MAE–well under a minute–translates to near-perfect accuracy, especially given that users typically request job durations in minutes or hours (Table 4). Additionally, we obtained MAE values of 2.7 % for CPU utilization and less than 1 % for GPU utilization. Interestingly, GPU utilization predictions yielded a higher R^2 score compared to those of CPU utilization.

Polaris - AWS prediction results are also very similar (Fig. 10) with a better MAE value of 14 seconds but a worse MAPE value of 41 %. The R^2 score values were similar ranging between 0.95-0.99. Although a MAPE value of 31–41 % might seem high, it is much less than the user-estimated runtime MAPE values of the Parallel Workloads Archive [25]. Table 5 shows MAPE rates for the user-requested runtime vs. the actual runtime of jobs collected from workload scheduler logs. Only ANL Intrepids error rate (10.96 %) is lower than ours, while the average MAPE of the 11 systems is 2057.81 %. Also, MAPE is a metric quite prone to outliers. Even a few outliers can drag the average percentage down a lot. In addition, we believe another factor affects this high value: short jobs. In our workloads, there are a lot of jobs with less than a minute or a few minutes of runtime. In this case, even a few seconds/minutes of difference between the actual and predicted runtime can cause a large percentage error. However, Service-level Agreements (SLAs) won't be affected if we add the margin error in MAPE or MAE into the predicted runtime when we make requests. The costs should be calculated taking the margin error into consideration as well.

We believe our model can be applied not only to integrated cloud and HPC environments but also to heterogeneous systems with varying node architectures. To evaluate this, we tested the model on the Polaris-IC2 case. Although both systems have similar CPU and GPU architectures, they differ significantly in terms of quantity, interconnects, and memory layout. Fig. 11 presents scatter plots comparing actual and predicted values for runtime and CPU/GPU utilization, while Table 6 summarizes the corresponding evaluation metrics. Compared to the AWS cases, runtime predictions in the Polaris-IC2 scenario are less tightly clustered and exhibit higher error rates. While the R^2 score for CPU utilization remains consistent with earlier results, the model struggles to predict 0 % or very low utilization cases. Additionally, CPU utilization does not exceed 50 %, indicating limited scalability or saturation for certain jobs. GPU utilization predictions remain consistent with previous results. Despite these challenges, the runtime prediction still achieves a MAE of 21 seconds, demonstrating that our model remains feasible and effective–even in the context of heterogeneous HPC systems–with sub-minute prediction accuracy.

To evaluate our second workload generator architecture, we collected data from the AWS PC2 system using CloudWatch, focusing on the small and mixed workload types. Fig. 12 presents the prediction results for the IC2-AWS PC2 and Polaris-AWS PC2 scenarios. We report only the runtime and GPU utilization results, as the model was unable to accurately predict CPU utilization due to insufficient data points in the CloudWatch-generated JSON files. Upon investigating the source of the model's poor performance for CPU utilization, we found that many tasks lacked usable CPU utilization data. This was primarily due to CloudWatch's 5-minute metric reporting interval, which is too coarse to capture the behavior of short-running tasks accurately. As a result, the model underperformed in predicting CPU utilization.

Figs. 12.a and 12.b show the runtime prediction results for the IC2-AWS PC2 and Polaris-AWS PC2 cases, respectively. As shown in Table 7, although the Polaris case yields lower MAE and MSE values, the IC2 case

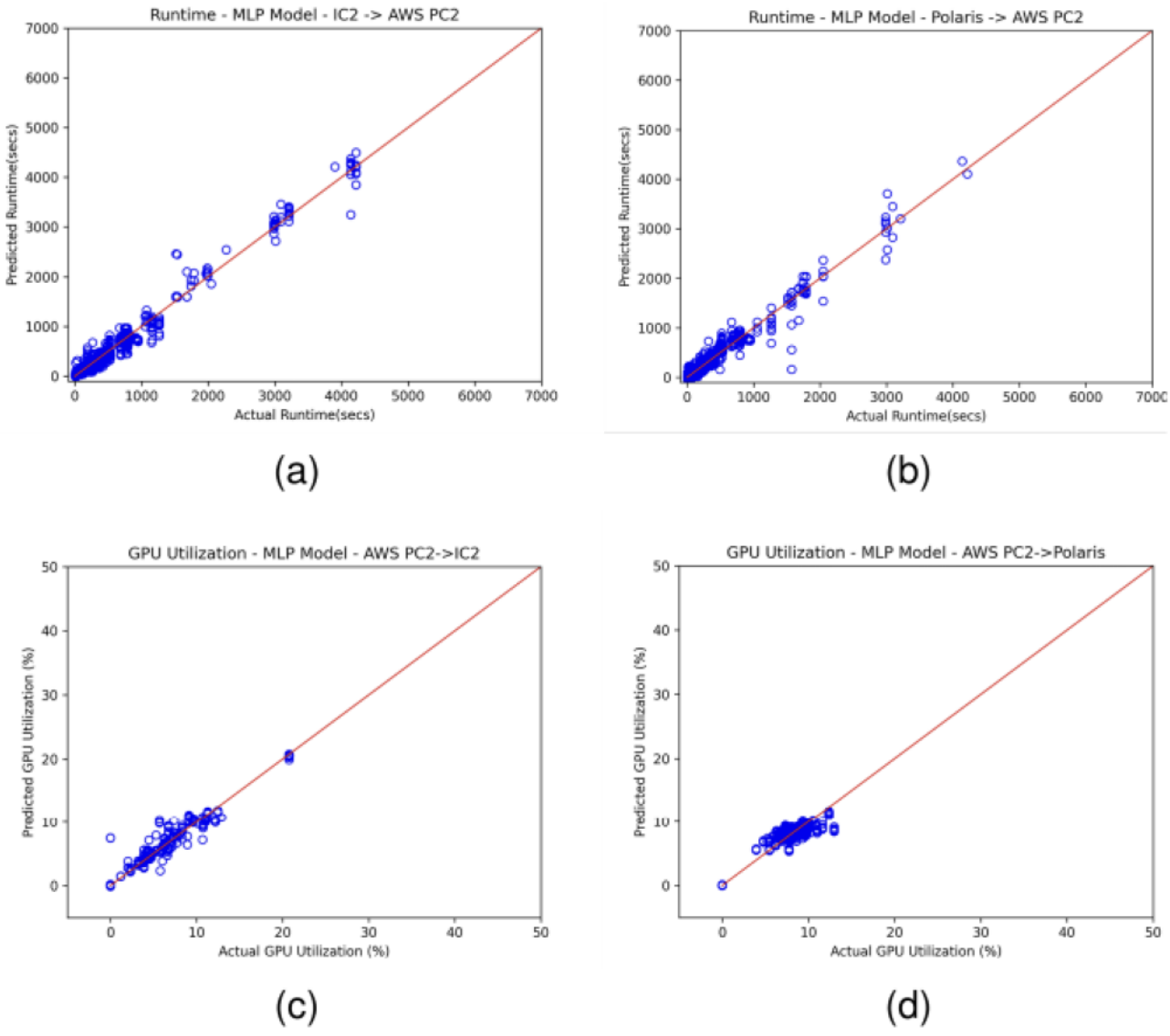

Fig. 12. Prediction results using CloudWatch logs - systems: IC2, polaris, AWS PC 2.

Table 4
Evaluation results - integrated HPC and cloud systems.

	IC2 - AWS PC 1			Polaris - AWS PC 1		
Metric	Runtime(secs)	CPU Util.(%)	GPU Util.(%)	Runtime (secs)	CPU Util.(%)	GPU Util.(%)
MAE	17.796	2.716	**0.069**	**14.858**	**2.472**	0.097
MSE	1551.831	20.734	0.089	**919.767**	**19.664**	**0.075**
R^2	**0.995**	**0.952**	0.990	0.993	0.951	**0.995**
MAPE	**31.289**			41.259		

performs better in terms of MAPE and R^2 score. The scatter plots also reveal more outliers for the Polaris case, with prediction points deviating further from the 45-degree ground truth line. When compared to RP-based monitoring, the CloudWatch-based predictions achieve comparable MAPE results but generally underperform in terms of MAE, MSE, and R^2 score. Figs. 12.c and 12.d show the GPU utilization prediction results for the IC2 and Polaris cases when jobs are transferred from the AWS PC2 system to the respective HPC systems. The scatter plot for the IC2 case indicates GPU utilization values reaching up to approximately 20 %, while the Polaris case remains below 15 %. This difference may be attributed to the random splitting of training and testing datasets.

As shown in Table 7, the Polaris case exhibits higher error rates compared to the IC2 case, although it achieves a better R^2 score. All results presented are based on the best-performing model among ten training runs, each initialized with different random weights and train/test splits.

Overall, the prediction results obtained using RP monitoring are superior to those achieved with CloudWatch monitoring. Several factors may contribute to this difference, including dataset size, monitoring frequency, and architectural variations in the underlying systems. In future work, we plan to extend our datasets to include additional experiments to further validate and improve the robustness of our model.

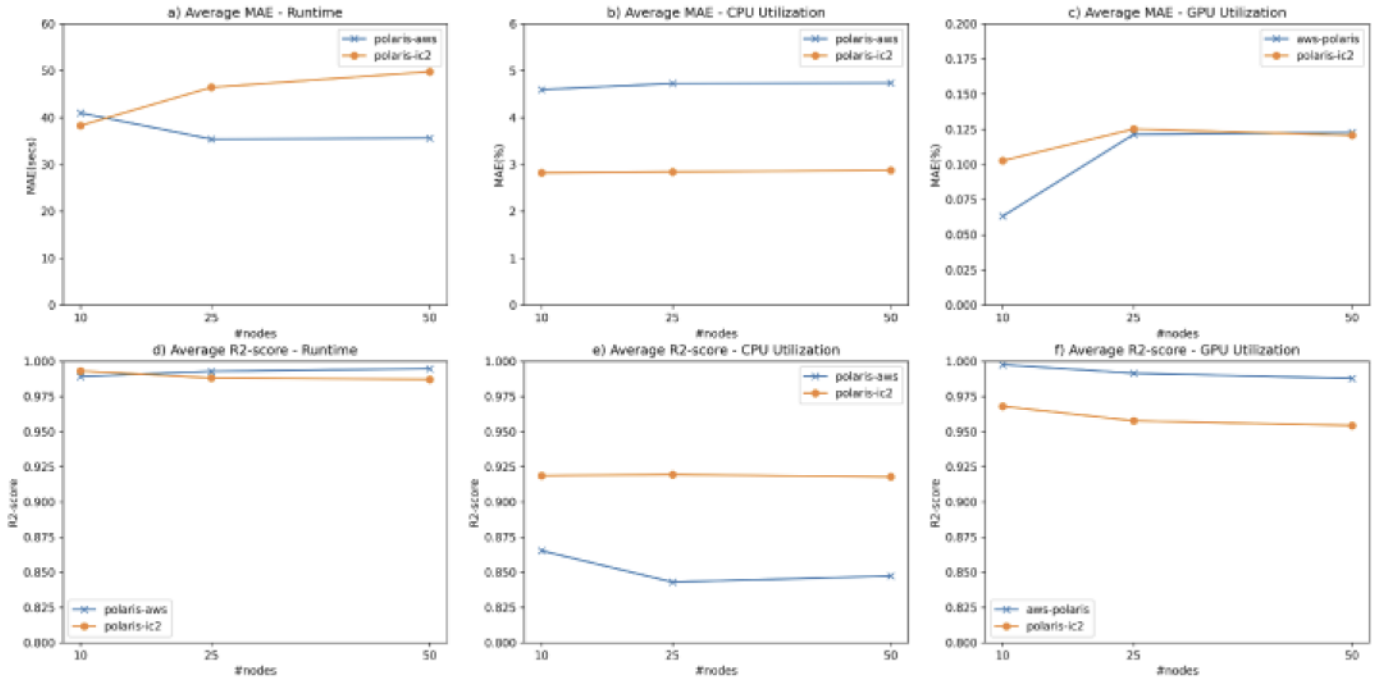

Fig. 13. Scalability vs. prediction accuracy.

Table 5
Parallel workloads archive user runtime estimates.

System	MAPE
ANL-Intrepid-2009	10.96
CEA-Curie-2011	3360.24
CIEMAT-Euler-2008	4438.84
CTC-SP2-1996	156.51
HPC2N-2002	2218.82
METACENTRUM-2013	7254.77
RICC-2010	2906.72
SDSC-BLUE-2000	47.99
SDSC-SP2-1998	75.50
UniLu-Gaia-2014	1680.95
lublin-1024	484.57
AVERAGE	**2057.81**

Table 6
Evaluation results - heterogeneous HPC systems.

	Polaris - IC2		
Metric	Runtime(secs)	CPU Util.(%)	GPU Util.(%)
MAE	21.786	1.888	0.079
MSE	1654.875	11.967	0.104
R^2	0.987	0.955	0.984
MAPE	396.311		

Table 7
Evaluation results using CloudWatch logs.

	IC2-AWS PC2		Polaris-AWS PC2	
Metric	Runtime(secs)	GPU Util.(%)	Runtime(secs)	GPU Util.(%)
MAE	38.474	0.087	36.347	0.164
MSE	5369.687	0.171	4949.583	0.230
R^2	0.985	0.975	0.971	0.981
MAPE	34.266		40.665	

Table 8
Comparison of average evaluation results of all models.

	LIN. REGR.	MLP	CNN	RNN	LSTM
Runtime					
MAE	125.96	**25.64**	86.81	253.14	219.16
MSE	49797.46	**2940.60**	27966.46	201781.71	189076.93
MAPE	734.99	**262.86**	761.52	1564.51	1499.36
R^2	0.68	**0.98**	0.81	−0.01	−0.09
CPU Utilization					
MAE	11.38	**3.29**	6.27	15.62	13.16
MSE	263.76	**34.90**	86.37	368.75	307.29
R^2	0.30	**0.91**	0.77	0.00	0.17
GPU Utilization					
MAE	0.64	**0.10**	0.39	2.33	1.89
MSE	2.17	**0.13**	0.89	10.39	11.50
R^2	0.74	**0.99**	0.92	−0.01	−0.28

4.4. Comparison to other models

We tested CNNs, RNNs, LSTMs, and Linear Regression prediction models in addition to our MLP model but none of the other models performed well against MLP. In Table 8, we present the comparison results for average MAE, MSE, MAPE, and R^2 scores for 10 runs with random weight initialization and train/test dataset split configuration. The results are then also averaged for all system pairs (IC2-AWS, Polaris-AWS, Polaris-IC2). While MLP is the clear winner in all evaluation metric categories, CNN is the runner up and Linear Regression follows it. RNN and LSTM perform the worst. Although LSTM results look better compared to RNN, RNN results were more stable across 10 runs. LSTMs had 1–2 outliers which were removed from the average. Overall, we can say that MLP is the best model while RNN/LSTM are the worst.

4.5. Scalability vs. prediction accuracy

We conducted a series of experiments to examine the impact of system scalability on the prediction accuracy of our model. Specifically, we increased the number of allocated nodes on the Polaris system to 10, 25, and 50 nodes and executed mixed workloads consisting of 1000 tasks. Fig. 13 presents the average MAE and R^2 score across 10 runs

Table 9
Comparison of related work.

Authors	Optimization	Runtime prediction	Sched./Pred. alg.	Input data	Applicability	System arch.	Max R^2 score
Fan et al. 2021 [3]	Scheduler	User Estimated	Reinforcement Learning	Trace Logs	Generic	Homogeneous	-
Mualem et al. 2001 [26]	Scheduler	User Estimated	FCFS + Backfilling	Trace Logs	Generic	Homogeneous	-
Grandl et al. 2014 [27]	Scheduler	User Estimated	Bin-packing	Monitoring	Specific to MapReduce	Homogeneous	-
Jojoo et al. 2023 [29]	Runtime	Predicted	Probabilistic Models	Sampling	Specific to MapReduce	Homogeneous	-
Fan et al. 2019[30]	Scheduler	User Estimated	Genetic Algorithms	Trace Logs	Generic	Homogeneous	-
Zhang et al. 2020[2]	Scheduler	User Estimated	Reinforcement Learning	Trace Logs	Generic	Homogeneous	-
Tanash et al. 2019[31]	Runtime + Memory	Predicted	Machine Learning	Trace Logs	Generic	Homogeneous	0.63
Chen et al. 2020 [32]	Runtime	Predicted	Machine Learning	Trace Logs	Generic	Homogeneous	0.98
Tanash et al. 2021 [33]	Runtime	Predicted	Machine Learning	Trace Logs	Generic	Homogeneous	0.77
Menear & Duplyakin 2023 [34]	Runtime	Predicted	Machine Learning	Trace Logs + Application	Generic	Homogeneous	0.47
Lamar et al. 2023 [35]	Runtime	Predicted	Machine Learning	Trace Logs + Input Params.	Specific to app.	Homogeneous	0.99
Naghshnejad 2018 [37]	Runtime	Predicted	State-space model	Trace Logs	Generic	Homogeneous	-
Yang et al. 2023 [36]	Runtime	Predicted	Machine Learning	Trace Logs + App. Path	Generic	Homogeneous	-
Wyatt II et al. 2018 [1]	Runtime + IO	Predicted	Neural Networks	Trace Logs	Generic	Homogeneous	-
Yildirim et al. 2025	Runtime + cpu/gpu util.	Predicted	Neural Networks	Monitoring	Generic	Integrated	**0.99**

of the prediction model, each using random train/test data splits and weight initialization. Our analysis revealed that the runtime prediction accuracy exhibits varying trends depending on the target system as we increase the number of nodes on the first system. While Polaris-AWS runtime MAE decreases slightly (Fig. 13.a), Polaris-IC2 runtime MAE increases. Their R^2 scores show a similar trend except that as MAE increases, R^2 decreases and vice versa (Fig. 13.d). However, the variations in R^2 are relatively minor and can be considered negligible. Increasing the number of nodes on the source system does not significantly affect the MAE of CPU utilization predictions (Fig. 13.b). However, a slight decrease in R^2 score is observed for the Polaris-AWS case as the number of nodes increases from 10 to 25 (Fig. 13.e). In contrast, the MAE value for GPU utilization prediction increases as we increase the number of nodes from 10 to 25 and R^2 scores show a trend that supports this phenomenon (Figs. 13.c and .f). Overall, the changes in the MAE and R^2 scores are very small. So, we conclude that the scalability differences of the two systems do not affect the prediction accuracy of our model.

5. Related work

Research on HPC scheduling optimization typically follows two main approaches: proposing new scheduling strategies based on user-estimated runtimes, or developing prediction models to estimate runtime and resource utilization. Table 9 presents a comprehensive summary of related studies, including our work, and categorizes them based on key characteristics. Due to the varying optimization objectives and data types used in these studies, direct comparisons between our model and prior work are challenging. Nevertheless, we compare the R^2 scores of our model against those of studies that reported this metric. First Come First Served (FCFS) with EASY backfilling is one of the earliest scheduling algorithms proposed, yet it remains the most widely used approach in current production HPC systems. The study in [26] compares two backfilling strategies–conservative and EASY–and concludes that, overall, their performance is comparable. Grandl et al. [27] propose a data collection strategy similar to ours, but their architecture is designed for clusters executing MapReduce jobs [28]. They assume homogeneous resource utilization across tasks within a job and demonstrate that monitoring past tasks can help predict future task utilization. Similarly, Jajoo et al. [29] predict job runtimes of MapReduce workloads using sampling tasks. Their approach addresses the limitations of historical data, particularly when job runtimes vary due to hardware or software changes. However, their method applies only to DAG-based scheduling and is not suitable for single-task jobs. Moreover, their evaluations are limited to homogeneous architectures, and they identify extending to heterogeneous systems as future work. While Fan et al. [30] adopt a genetic algorithm for scheduling, more recent studies have increasingly favored reinforcement learning techniques–especially the actor-critic model combined with neural networks–to enable intelligent scheduling decisions [2,3]. It is worth noting that all of these approaches are designed for homogeneous systems, rely on user-estimated runtimes, and–with the exception of Grandl et al. and Jajoo et al.–primarily use scheduler trace logs in their models.

The second category of related work–including our own–focuses on predicting job runtime and, in some cases, resource utilization to support more informed scheduling decisions. The studies in [31–33] are closely related, employing trace logs and traditional machine learning techniques such as random forests, along with ensemble learning models like LightGBM. Some works incorporate application-level features to enhance model accuracy [34,35]. For example, Lamar et al. [35] collect data from five specific physics applications executed with varying input parameters. Their model achieves an impressive R^2 score of 0.99, but only for a single application type. While this demonstrates high accuracy, it also limits generalizability–the model cannot be applied to applications outside of the training set. Yang et al. [36] suggest that applications with similar file paths tend to have similar execution times, while Wyatt et al. [1] utilize neural networks to predict both runtime and I/O usage. Like those in the first category, these studies are also designed for homogeneous systems and primarily rely on scheduler trace

logs for training data. In contrast, our work presents a generic model capable of accurately predicting runtime across a variety of applications and systems, including integrated cloud and HPC environments. Our model achieves the highest reported R^2 score among the studies that provide this metric. The most comparable work in terms of prediction accuracy is by Chen et al. [32], who report a similar R^2 score–but only for a single system. For other systems, their model performance drops to an R^2 range of 0.80-0.81. To the best of our knowledge, ours is the only model designed to operate effectively in integrated and heterogeneous computing environments.

6. Conclusion

By leveraging system resource utilization data and HPC job characteristics, our neural network model accurately predicts job runtime and CPU/GPU usage within error margins of under 1 minute and 3 %, respectively. This prediction framework operates across integrated cloud-HPC environments using a monitoring strategy that captures load metrics from both systems. When integrated into workload managers like RP or schedulers like Slurm, it can enable informed, cost-aware decisions for job migration to cloud platforms based on accurate runtime and instance specifications. Future work will focus on scaling the dataset to encompass larger HPC and cloud systems, as well as longer-duration jobs, to enhance model generalizability. We plan to integrate the prediction model into a scheduling framework that utilizes runtime estimates for job placement and migration decisions. To assess the applicability of transfer learning, the model will be retrained and fine-tuned using heterogeneous datasets collected from additional platforms, such as Google Cloud. Furthermore, we will investigate advanced data transfer optimization strategies to minimize staging latency during data movement from HPC storage to cloud-based environments.

CRediT authorship contribution statement

Esma Yildirim: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Project administration, Methodology, Funding acquisition, Data curation, Conceptualization; **Mohab Hussein:** Writing – review & editing, Writing – original draft, Software, Data curation; **Mikhail Titov:** Writing – review & editing, Writing – original draft, Supervision, Resources, Project administration, Methodology, Investigation, Conceptualization; **Ozgur Ozan Kilic:** Writing – review & editing, Writing – original draft, Software, Resources, Methodology, Investigation, Conceptualization.

7. Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used ChatGPT in order to improve language and readability. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the publication.

Data availability

We have provided the link to the data in the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper. The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Esma Yildirim reports financial support was provided by National Science Foundation. Esma Yildirim reports financial support was provided by US Department of Energy. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This project was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Visiting Faculty Program (VFP) and by the National Science Foundation under grant #2100027 "Using Cloud Technologies to Develop the Data Analysis Skills of Community College Students". This research used computing resources from the Scientific Data and Computing Center (SDCC) at Brookhaven National Laboratory, which is sponsored by the U.S. Department of Energy, Office of Science, under Contract No. DE-SC0012704. This research also used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

[1] M.R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D.H. Ahn, M. Taufer, Prionn: predicting runtime and io using neural networks, in: Proceedings of the 47Th International Conference on Parallel Processing, 2018, pp. 1–12. https://doi.org/10.1145/3225058.3225091

[2] D. Zhang, D. Dai, Y. He, F.S. Bao, B. Xie, RLScheduler: An automated HPC batch job scheduler using reinforcement learning, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–15. https://doi.org/10.1109/SC41405.2020.00035

[3] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, M.E. Papka, Deep reinforcement agent for scheduling in HPC, in: 2021Del InsThinspace IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2021, pp. 807–816. https://doi.org/10.1109/IPDPS49936.2021.00090

[4] E. Yildirim, A. Akon, Predicting short-Term variations in end-to-End cloud data transfer throughput using neural networks, IEEE Access 11 (2023) 78656–78670. https://doi.org/10.1109/ACCESS.2023.3299311

[5] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (5) (1989) 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[6] A.A. Movassagh, J.A. Alzubi, M. Gheisari, D. Rezania, S. Malekian, Artificial neural networks training algorithm integrating invasive weed optimization with differential evolutionary model, J. Ambient Intell. Humaniz. Comput. 14 (2023) 6017–6025. https://doi.org/10.1007/s12652-020-02623-6

[7] The NAS Parallel Benchmarks, 2024, https://www.nas.nasa.gov/software/npb.HTML. (accessed July 18, 2025).

[8] F. Chollet, Image Classification from Scratch, 2024, https://keras.io/examples/vision/image_classification_from_scratch/. (accessed July 18, 2025).

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., {TensorFlow}: A system for {Large-Scale} machine learning, in: 12Th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265–283. https://doi.org/10.5555/3026877.3026899

[10] D.W. Walker, J.J. Dongarra, MPI: A standard message passing interface, Supercomputer 12 (1996) 56–68. https://www.researchgate.net/publication/2809665_Mpi_A_Standard_Message_Passing_Interface.

[11] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55. https://doi.org/10.1109/99.660313

[12] Z. Jiang, W. Gao, L. Wang, X. Xiong, Y. Zhang, X. Wen, C. Luo, H. Ye, X. Lu, Y. Zhang, et al., HPC AI500: A benchmark suite for HPC AI systems, in: Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium, Bench 2018, Seattle, WA, USA, December 10Del-Ins-13, 2018, Revised Selected Papers 1, Springer, 2019, pp. 10–22. https://doi.org/10.1007/978-3-030-32813-9_2

[13] P. Microsoft, Kaggle: Cats vs. Dogs Dataset [dataset], 2024, https://www.microsoft.com/en-us/download/details.aspx?id=54765. Microsoft, v1.0.

[14] F. Chollet, Xception: deep learning with depthwise separable convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 1251–1258. https://doi.org/10.1109/CVPR.2017.195

[15] A.B. Yoo, M.A. Jette, M. Grondona, Slurm: simple linux utility for resource management, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2003, pp. 44–60. https://doi.org/10.1007/10968987_3

[16] D.G. Feitelson, D. Tsafrir, D. Krakov, Experience with using the parallel workloads archive, J. Parallel Distrib. Comput. 74 (10) (2014) 2967–2982. https://doi.org/10.1016/j.jpdc.2014.06.013

[17] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, J. Wilkes, Large-scale cluster management at google with borg, in: Proceedings of the Tenth European Conference on Computer Systems, 2015, pp. 1–17. https://doi.org/10.1145/2741948.27419

[18] A. Iosup, H. Li, M. Jan, S. Anoep, C. Dumitrescu, L. Wolters, D.H.J. Epema, The grid workloads archive, Future Gen. Comput. Syst. 24 (7) (2008) 672–686. 10.1016/j.future.2008.02.003.

[19] A. Merzky, M. Turilli, M. Titov, A. Al-Saadi, S. Jha, Design and performance characterization of radical-pilot on leadership-class platforms, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 818–829. https://doi.org/10.1109/TPDS.2021.3105994

[20] Amazon CloudWatch, 2024, https://aws.amazon.com/cloudwatch/. (accessed July 18, 2025).

[21] E. Yildirim, M. Hussein, M. Titov, UNN-Pred-CloudHPC [software], GitHub, 2024, https://github.com/esmayildirim/UNN-Pred-CloudHPC-Public.git.

[22] R.L. Henderson, Job scheduling under the portable batch system, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1995, pp. 279–294. https://doi.org/10.1007/3-540-60153-8_34

[23] U. Lublin, D.G. Feitelson, The workload on parallel supercomputers: modeling the characteristics of rigid jobs, J. Parallel Distrib. Comput. 63 (11) (2003) 1105–1122. https://doi.org/10.1016/S0743-7315(03)00108-4

[24] D.G. Feitelson, Metrics for parallel job scheduling and their convergence, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2001, pp. 188–205. https://doi.org/10.1007/3-540-45540-X_11

[25] Parallel Workloads Archive, 2024, https://www.cs.huji.ac.il/labs/parallel/workload/logs.HTML. (accessed July 18, 2025).

[26] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Trans. Parallel Distrib. Syst. 12 (6) (2001) 529–543. https://doi.org/10.1109/71.932708

[27] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, A. Akella, Multi-resource packing for cluster schedulers, ACM SIGCOMM Comput. Commun. Rev. 44 (4) (2014) 455–466. https://doi.org/10.1145/2740070.2626334

[28] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. https://doi.org/10.1145/1327452.132749

[29] A. Jajoo, Y.C. Hu, X. Lin, N. Deng, SLearn: A case for task sampling based learning for cluster job scheduling, IEEE Trans. Cloud Comput. 11 (3) (2023) 2664–2680. https://doi.org/10.1109/TCC.2022.3222649

[30] Y. Fan, Z. Lan, P. Rich, W.E. Allcock, M.E. Papka, B. Austin, D. Paul, Scheduling beyond CPUs for HPC, in: Proceedings of the 28Th International Symposium on High-Performance Parallel and Distributed Computing, 2019, pp. 97–108. https://doi.org/10.1145/3307681.3325401

[31] M. Tanash, B. Dunn, D. Andresen, W. Hsu, H. Yang, A. Okanlawon, Improving HPC system performance by predicting job resources via supervised machine learning, in: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), 2019, pp. 1–8. https://doi.org/10.1145/3332186.3333041

[32] X. Chen, H. Zhang, H. Bai, C. Yang, X. Zhao, B. Li, Runtime prediction of high-performance computing jobs based on ensemble learning, in: Proceedings of the 2020 4Th International Conference on High Performance Compilation, Computing and Communications, 2020, pp. 56–62. https://doi.org/10.1145/3407947.3407968

[33] M. Tanash, H. Yang, D. Andresen, W. Hsu, Ensemble prediction of job resources to improve system performance for slurm-based hpc systems, in: Practice and Experience in Advanced Research Computing, 2021, pp. 1–8. https://doi.org/10.1145/3437359.3465574

[34] K. Menear, D. Duplyakin, Is knowledge about running applications helping improve runtime prediction of HPC jobs? in: Practice and Experience in Advanced Research Computing, 2023, pp. 463–465. https://doi.org/10.1145/3569951.3597558

[35] K. Lamar, A. Goponenko, O. Aaziz, B.A. Allan, J.M. Brandt, D. Dechev, Evaluating hpc job run time predictions using application input parameters, in: Proceedings of the 17Th ACM International Conference on Distributed and Event-based Systems, 2023, pp. 127–138. https://doi.org/10.1145/3583678.3596893

[36] W. Yang, X. Liao, D. Dong, J. Yu, Exploring job running path to predict runtime on multiple production supercomputers, J. Parallel Distrib. Comput. 175 (2023) 109–120. https://doi.org/10.1016/j.jpdc.2023.01.001

[37] M. Naghshnejad, M. Singhal, Adaptive online runtime prediction to improve HPC applications latency in cloud, in: 2018Del InsThinspace IEEE 11Th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 762–769. https://doi.org/10.1109/CLOUD.2018.00104