AI-Driven Resource Optimization for High-Performance Computing: A Comprehensive Framework

Candidacy Proposal

By

Manikya Swathi Vallabhajosyula

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2025

Candidacy Proposal Committee:

Rajiv Ramnath, Advisor

Byna Suren

Subramoni Hari

Panda Dhabaleswar

Abstract

High-performance computing (HPC) environments are essential for scientific research across domains like climate modeling, genomics, and deep learning. However, researchers face significant challenges in resource provisioning, leading to inefficiencies in job execution. These challenges include the complexity of resource selection, inconsistencies in job configurations, a steep learning curve for performance profiling tools, and the constant need to adapt to changing cyberinfrastructure (CI) policies. Existing resource estimators lack adaptability to diverse architectures and batch policies, forcing researchers to rely on manual adjustments, which leads to resource over-provisioning, increased queue wait times, and higher computational costs. This research introduces an **AI-driven framework for intelligent resource provisioning** in HPC environments, featuring two core components:

1. **HPC Application Resource Predictor (HARP):** A machine learning-based tool that generates application-specific resource estimates by analyzing code structure, parameter configurations, and hardware variability.

2. **Intelligent CI-Aware Scheduling:** A system that adapts resource estimates to CI constraints, considering budget limitations and scheduling policies to optimize job execution.

Key innovations include:

1. **Efficient Training Data Generation:** A downsized execution approach reduces data collection costs by a factor of 7.

2. **DNN-Estimator for AI Workloads:** A specialized predictor for deep neural networks (DNNs) that accounts for architecture, dataset size, and hardware configurations.

3. **Pre-SLURM Validation Database:** A structured database of HPC system configurations, integrating CI documentation and real-time system data.

4. **Intelligence Plane (IP):** A dynamic execution planning system that integrates with TAPIS to optimize scheduling, monitoring, and rescheduling of jobs.

To validate the framework, we apply it to an animal ecology use case, where AI inference models analyze sensor data at the edge, triggering automated model retraining pipelines. The Intelligence Plane orchestrates data transfer, labeling, and training, leveraging HARP and the DNN-Estimator to optimize resource allocations dynamically. The system continuously refines its execution estimates, ensuring efficient utilization of both edge and HPC resources. Future work will explore enhancements to the DNN-Estimator, including methods to improve training time predictions with reduced reliance on direct hardware execution. Additionally, we aim to refine user interaction through an integrated chatbot assistant in iScheduler, enabling researchers to explore configurations and receive real-time resource selection guidance. Furthermore, we will focus on showcasing the generalizability of our framework's features across diverse use cases, demonstrating its adaptability to different scientific domains and computational workflows.

Vita

2013 ..... B.Tech Computer Science, VN-RVJIET, Hyderabad, India

2018 ..... M.S. Computer Science, University of Minnesota - Duluth, Minnesota, USA

Publications

Research Publications

M. S. Vallabhajosyula and R. Ramnath "Towards Practical, Generalizable Machine-Learning Training Pipelines to Build Regression Models for Predicting Application Resource Needs on HPC Systems". *Practice and Experience in Advanced Research Computing 2022: Revolutionary: Computing, Connections, You*, pp. 1–5, 2022.

S. Vallabhajosyula and R. Ramnath "Establishing a Generalizable Framework for Generating Cost-Aware Training Data and Building Unique Context-Aware Walltime Prediction Regression Models". *2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 497–506, 2022.

M. S. Vallabhajosyula and R. Ramnath "Towards Characterizing DNNs to Estimate Training Time using HARP (HPC Application Resource (runtime) Predictor)". *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, pp. 483–485, 2023.

M. S. Vallabhajosyula, S. S. Budhya, and R. Ramnath "Reference Implementation of Smart Scheduler: A CI-Aware, AI-Driven Scheduling Framework for HPC Workloads". *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, pp. 1–4, 2024.

Conference Posters & Presentations

M. S. Vallabhajosyula and R. Ramnath "Towards Characterizing DNNs to Estimate Training Time using HARP (HPC Application Resource (runtime) Predictor)". *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, pp. 483–485, 2023.

M. S. Vallabhajosyula, S. S. Budhya, A. Jain, M. Baig, and R. Ramnath "Orchestrating a DNN Training Job Using an iScheduler Framework: A Use Case". *Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, pp. 1–3, 2024.

Fields of Study

Major Field: Artificial Intelligence for High Performance Computing Optimization

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

High-performance computing (HPC) environments, such as those at the Ohio Supercomputer Center (OSC) and the Texas Advanced Computing Center (TACC), are essential for running computationally intensive tasks like weather simulations, genome sequencing, and deep neural network (DNN) training. These shared-resource systems allow multiple users to run jobs with varying resource demands and execution times. While HPC platforms offer different interfaces for resource requests, efficient utilization depends on users' ability to assess workload characteristics and request appropriate resources accurately. However, managing jobs in HPC environments presents several challenges, often leading to inefficient resource allocation. Users must navigate complex configurations, adapt to evolving infrastructure, and work with limited estimation tools, frequently requiring manual adjustments. Continuous monitoring and repeated job resubmissions add to inefficiencies, extending execution times. These challenges include:

- Resource Complexity: Researchers need a deep understanding of available resources, allocation policies, limitations, costs, and configuration options to utilize HPC resources effectively.

- Inconsistent Job Configurations: Simple estimations become unreliable due to changes in code, parameter settings, software, or hardware upgrades.

- Steep Learning Curve: Navigating profiling and analytics tools to understand job requirements and optimize resource allocations requires significant training and effort.

- Constant Updates: Researchers must stay informed about cyberinfrastructure (CI) and middleware changes to maintain optimal resource utilization.

- Limited Resource Estimators: Current tools do not generate execution plans that account for available architectures or CI batch policies, necessitating manual adjustments.

- Continuous Monitoring: Researchers must constantly monitor progress and performance despite pre-profiling jobs, often needing to resubmit jobs to improve efficiency or extend wall time.

- Human Intervention: The frequent need to separate jobs into smaller jobs to fit CI allocation policies leads to a longer time to completion due to the need for human monitoring and intervention.

These challenges often result in researchers over-provisioning resources to ensure job completion, leading to longer queue wait times, inefficient resource use, and increased costs. Chapter 2 describes the motivation for creating AI-enabled frameworks to address these challenges.

This dissertation explores and presents the integration of machine learning into HPC cyberinfrastructure to enhance resource provisioning. AI-driven solutions can

help users better understand their resource requirements, enabling more informed allocation decisions and improving overall efficiency. We aim to address resource management challenges in HPC environments by developing intuitive and programmable AI-based methods.

Our research introduces a comprehensive framework that utilizes two key strategies:

- Application-Specific Resource Estimation: We developed the HPC Application Resource Predictor (HARP). This advanced tool leverages machine learning to create tailored models for estimating the resource requirements of individual applications. HARP addresses the complexity of modern HPC applications by considering diverse factors such as application configurations, input size, and hardware variability. This approach significantly improves traditional estimation methods by adapting models to each application's unique characteristics. The reference architecture and the components used to create HARP are described in Chapter 4 Section 4.1.

- Intelligent CI-Aware Scheduling: This strategy produces application-specific estimates made specific to Cyberinfrastructure (CI) configurations. Doing so enables context-aware execution recommendations across diverse HPC systems that are feasible based on the CI budget limitations and costing policies. This intelligent scheduling system considers the estimated resource requirements and the current state of the HPC environment in terms of available resources to schedule or reschedule jobs. The reference architecture and the components used to create the Intelligent Scheduler are described in Chapter 4 Section 4.2.

Key innovations of our research include:

- Cyberibnfrastructre Configuration Database: We constructed a comprehensive database of HPC system configurations by combining manually scraped CI documentation with programmatically generated information. This database is crucial in assessing resource suitability and generating viable execution plans. We call this database a pre-SLURM validation database. Chapter 4 explains the development of a database that supplies data to the estimation engine, assists in generating execution plans, and evaluates the feasibility of executions.

- Efficient Training Data Generation: We developed a novel method to generate application-specific training data through downsized execution campaigns. This approach reduces the cost and time required for data collection by a factor of 7, making it feasible to create accurate models for a wide range of applications. Chapter 5 describes how we emulate application profiling to generate scalable training datasets for resource estimations.

- DNN-Estimator for AI Workloads: Recognizing the growing importance of deep learning in scientific research, we created a specialized estimator for deep neural network (DNN) training and inferencing tasks. The DNN-Estimator can predict resource requirements based on network architecture, dataset characteristics, and hardware configurations. Chapter 8 describes the white box-model aware resource estimators for training the model.

- Intelligence Plane: This component dynamically generates execution plans using the pre-SLURM database and resource estimators. Integrated with TAPIS, it can schedule, monitor, and reschedule jobs in real-time, adapting to changing

system conditions and job performance. The reference architecture and the components used to create the Intelligent Scheduler are described in Chapter 4.

To demonstrate the real-world applicability of our framework, we present a detailed use case in animal ecology primarily focused on workflow interactions with centers like OSC or TACC. This pipeline begins with data collection field sensors or cameras and is analyzed by AI inference models deployed on edge, triggering a chain of automated processes for model retraining for better accuracy. Once new data is transferred to the center, the Intelligence Plane (IP) orchestrates tasks, such as data labeling and model retraining. The IP employs sophisticated resource estimation techniques, using the DNN Estimator or HARP, to determine optimal resource allocation. It then validates these requirements against a pre-SLURM database and generates execution plans accordingly. The IP continuously monitors job progress, dynamically rescheduling tasks to ensure efficient resource utilization.

Our future work will proceed in two directions: 1. enhancing DNN estimator efficiency, and 2. improving user interaction through chatbot integration. Our DNN execution time estimation framework currently relies on generating High-Level Optimization (HLO) graphs for a given DNN model architecture (DNN model summary)—an intermediate representation used by compilers like XLA in machine learning frameworks—to predict training times. While HLO graphs effectively optimize computations for different hardware accelerators, our current approach faces a significant bottleneck: we must execute each new DNN model on target hardware platforms to generate these graphs before estimating training times. For example, estimating VGG16 training time on a V100 GPU requires running the model for one step on that specific hardware to generate the HLO graph, from which we extract features

for our time estimation model. This dependency necessitates maintaining short-lived queues across various systems just for graph generation. To overcome this limitation, we propose using generative AI models to dynamically create HLO graphs for given DNN architectures and hardware configurations, eliminating the need for actual hardware execution and significantly reducing computational overhead. Additionally, we intend to develop an interactive chatbot by integrating our iScheduler helper framework with an Ollama model. This chatbot will enable users to query models, explore configurations, and receive real-time feedback on resource selection and feasibility options. Essentially, we aim to streamline the resource estimation process and provide a more intuitive, chat-based assistant for resource management in HPC environments, thereby enhancing the user experience while improving system efficiency.

Chapter 2: Challenges in Efficient Utilization of HPC Resources

In this section, we provide a brief overview of our problem motivation by identifying common patterns across different workflows. Our goal is to understand the necessity of developing new software solutions centered around building machine learning-driven tools to enhance user interactions with HPC environments. Through our observations of a genome sequencing researcher's interaction with the OSC environment, we identified the following challenges:

- Researchers rely on batch scripts curated for their lab with assistance from an HPC expert at OSC. These scripts are distributed as templates but become outdated and are not tailored to individual researchers despite variations in workflows.

- Workflow differences arise based on factors such as the type of raw sequence data and the specific analysis tools required for assembly.

- Researchers often fail to optimize parallelization parameters, such as the -threads option in Trimmomatic. This option, introduced in later versions, is frequently overlooked because batch scripts were initially designed for older

versions. Even when used, -threads is often set arbitrarily (e.g., 10) instead of being tuned for optimal CPU allocation.

Similarly, while analyzing AI researchers submitting GPU-centric job requests, we observed the following inefficiencies:

- Many researchers request full-node allocations with all GPUs, even when they are not utilizing distributed training.
- They also request all available CPU cores while training on GPUs, without profiling their workloads to understand CPU utilization needs.

2.1 Selection of Representative HPC Workloads

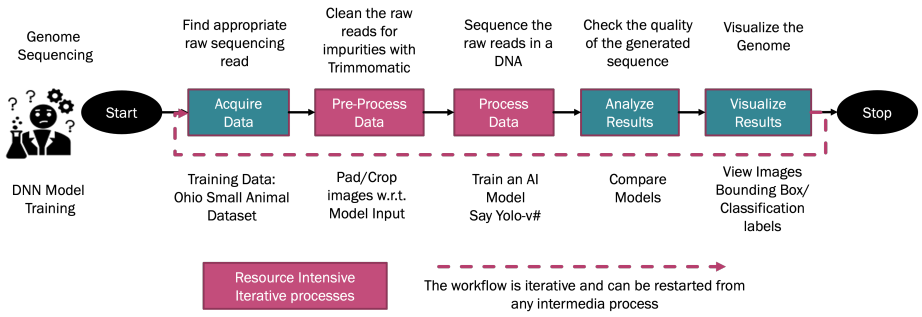


Figure 2.1: Overview of a Data-Driven Computational Workflow for Genome Assembly and DNN Training Pipelines

These three applications were chosen to represent different points in the HPC application spectrum, considering variations in compute intensity, input-output (I/O) demands, and memory requirements.

- Gray-Scott (Chemical Diffusion Simulator): This application simulates chemical diffusion based on Gray-Scott equations, using a 3D array to model the interaction of two reacting substances over a range of time steps. It takes parameters such as diffusion coefficients, reactant densities, feed/kill rates, and simulation steps. Since it performs intensive array manipulations to simulate diffusion dynamics, it is classified as compute-intensive.

- Trimmomatic (Genome Sequencing Preprocessing): Pre-cleaning is a critical step in genome sequencing and assembly to improve the quality of results. Trimmomatic is a widely used tool for removing impurities from Illumina short reads by trimming or filtering out low-quality bases while retaining high-quality ones. The application reads raw sequencing data, processes it, and writes the cleaned data back to storage. Due to its frequent read/write operations alongside computational processing, Trimmomatic is both compute- and I/O-intensive.

- Deep Neural Networks (DNNs) for Image Classification: We evaluated three large Convolutional Neural Network (CNN) models—VGG16, ResNet50, and InceptionV3—using the Keras API on TensorFlow. Each model contains millions of tunable parameters and requires loading the training data as RGB values into memory before initialization and training. Due to the high computational load and large memory footprint required for model execution, these applications are categorized as both compute- and memory-intensive.

This selection provided us a different spectrum of HPC workloads, covering compute-bound, I/O-bound, and memory-bound applications, enabling a diverse analysis of

resource interactions in high-performance environments. Figure 2.1 presents a general pipeline for data-driven computational workflows, while Figure 2.2 provides a mind map outlining various tools, datasets, and execution endpoints associated with different workflows

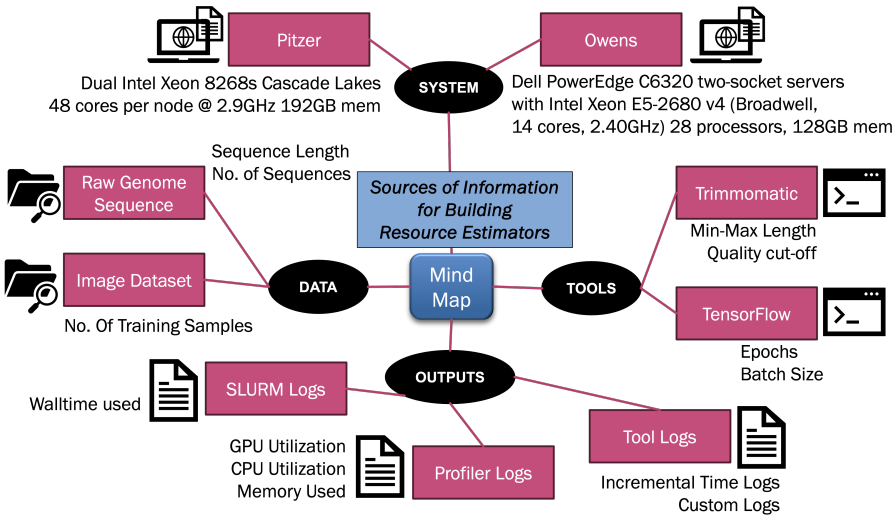


Figure 2.2: Mind Map of Data, Tools, and System Components for Genome Assembly and DNN Training Workflows

2.2 Challenges in HPC Resource Provisioning

Figure 2.3 illustrates the HPC job allocation and optimization workflow using SLURM and Grafana for job scheduling. The process begins with creating allocation scripts, which are either generated based on OSC Batch example documentation or adapted from peer-shared templates. Users typically analyze execution logs and Grafana metrics only when a job fails, diagnosing inefficiencies and adjusting resource

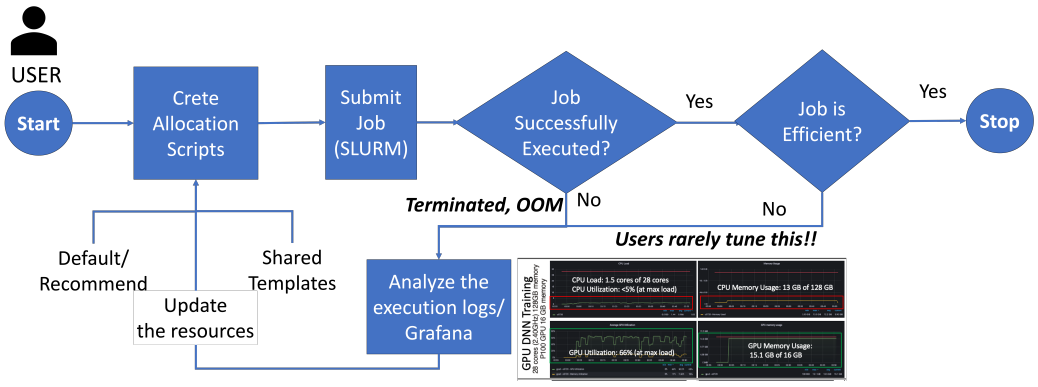


Figure 2.3: Workflow for Job Allocation, Execution, and Optimization in HPC Environments

allocations accordingly. However, resource fine-tuning is rarely performed for successfully executed jobs, even when it could reduce execution costs or improve scalability (as shown in Figure 2.4). Job efficiency is assessed by monitoring CPU, memory, and GPU utilization using tools such as Grafana or XDMoD.

During these workloads, HPC users—regardless of their expertise with supercomputers—face several common challenges, including:

- Job Termination Due to Resource Under-Provisioning: If a job exceeds the allocated walltime, it gets terminated, and without checkpointing, all progress is lost, leading to costly reruns. SLURM doesn't allow general users to extend their allocation during job execution.

- Out-of-Memory (OOM) Errors for DNN Training on GPUs: Training a deep neural network (DNN), such as Inception V3, with a large batch size (e.g., 256) on a P100 GPU (which has 15.1 GB of usable memory out of 16 GB) can exceed memory limits, causing the job to fail.

- Difficulty in Fine-Tuning CPU Allocation for GPU-Centric Workloads: Users struggle to determine the optimal CPU allocation needed for data preprocessing during GPU training without impacting overall training time.

- Lack of Understanding in Interpreting Grafana Logs for CI Policies: Users face challenges in tuning –ntasks-per-node (no. of CPU cores) based on CPU memory requirements. For example, on Owens (where max memory per core is 4.214 GB), a workflow requiring 30 GB of memory should request at least 8 CPU cores to accommodate the workload efficiently.

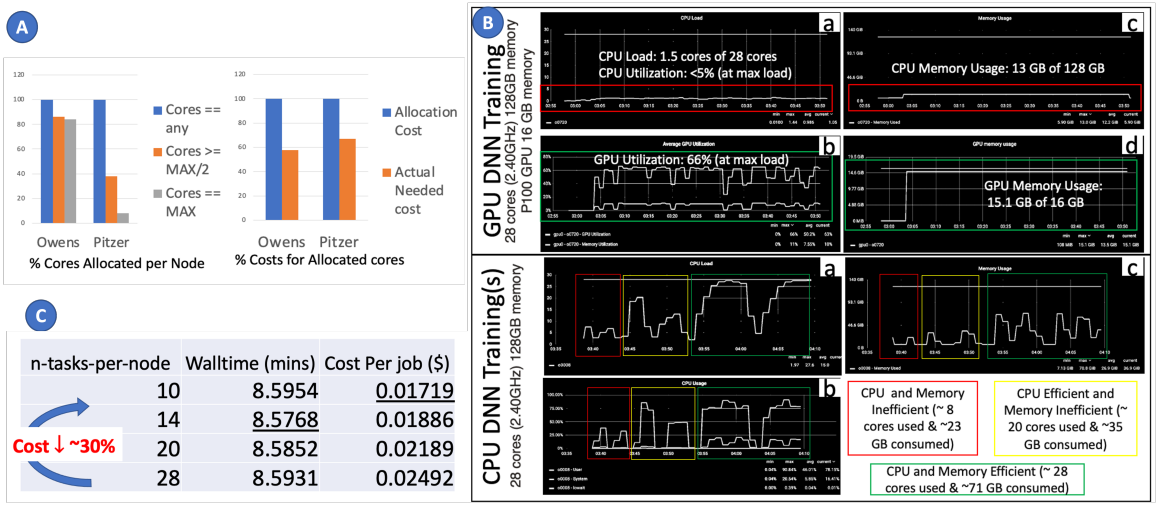
- Policy Variations in Cyber-Infrastructure (CI) Allocations:

  – At OSC, users can request partial nodes, enabling node sharing to accommodate multiple users simultaneously. When users submit job requirements (CPU cores, memory, GPUs, walltime) to SLURM, it dynamically assigns the job to an appropriate queue (e.g., serial, gpu-serial, hugemem, parallel, gpu-backfill).

  – At TACC, node sharing is not enabled, meaning users must explicitly request a queue that best fits their workload. For example, the rtx-dev queue is used for short 2-hour jobs, while the rtx queue is allocated for 1-day jobs.

- Need for Continuous Job Monitoring and Adaptive Scheduling: Users should actively monitor job execution to ensure it runs as expected. When necessary, jobs can be divided into smaller sub-jobs and scheduled either sequentially or strategically. This helps achieve faster allocation within the HPC batch queue

system based on availability and ensures the jobs fit within queue limitations, improving overall computational efficiency.



n-tasks-per-node	Walltime (mins)	Cost Per job ($)
10	8.5954	0.01719
14	8.5768	0.01886
20	8.5852	0.02189
28	8.5931	0.02492

Figure 2.4: A snapshot of CPU and GPU Resource Utilization, Allocation Inefficiencies, and Cost Optimization in HPC Workloads.

Figure 2.4 provides an overview of CPU and GPU resource utilization, allocation inefficiencies, and cost optimization for HPC workloads. (A) Core Allocation and Cost Analysis compares allocated versus required CPU cores on Owens and Pitzer supercomputers, highlighting disparities in resource allocation and the associated costs. The analysis demonstrates how over-provisioning leads to unnecessary expenses. (B) GPU DNN Training Metrics presents Grafana logs showcasing CPU load, GPU utilization, and memory consumption during deep neural network (DNN) training on a P100 GPU. It highlights inefficiencies in CPU allocation and illustrates how resource demands fluctuate based on DNN hyperparameters. (C) Cost Savings with Optimal CPU Allocation examines the impact of adjusting n-tasks-per-node to improve CPU

efficiency. The results indicate that reducing excess core allocation can lower costs by 30% without increasing wall time, demonstrating the importance of resource-aware job scheduling in HPC environments.

2.3 Key Research Questions

By analyzing **user-HPC interaction endpoints and workload patterns**, as outlined above, this thesis aims to develop **software and ML-based solutions** to enhance **resource allocation and utilization**. The proposed approaches address the following key research questions:

Question 1: Can AI models be used to develop resource estimators for predicting runtime and memory feasibility for user-specific jobs?

- **What datasets are available or needed** to train models that estimate job-specific resources? If none exist, how can they be efficiently created?
- **What features should be captured** to predict resource requirements, such as runtime or memory?
- **Which models** can best predict resource requirements? Can a **"one model fits all"** approach work across different workflows and estimations (e.g., memory, walltime)?
- **Can standard ML metrics** like **mean squared error (MSE)** be used to validate resource estimations, or do we need **custom evaluation metrics**?

Question 2: Can a framework be designed to integrate these estimators into a SLURM wrapper, enabling automated job submission, tracking, and execution based on resource availability and CI batch policy limitations?

- **Can we create a multi-purpose cyberinfrastructure database** that helps:
  - Users understand all **available queues** and their limitations?
  - Estimate the **cost of allocations** based on job configurations?
  - Assess **execution feasibility** and dynamically configure scheduling?
- **Can long-running jobs be tracked** using **SLURM or tool logs**? What ecosystem is required to monitor job progress and efficiency?

In Chapter 3, we examine existing solutions that address the aforementioned questions and identify the bottlenecks that serve as the driving factors for our research.

Chapter 3: Background: Resource Estimations and Schedulers

Data centers are among the largest energy consumers, accounting for approximately 4.4% of total U.S. electricity usage in 2023, with projections suggesting an increase to 6.7%–12% by 2028, effectively tripling energy demands within the next three years [1]. Despite the rising power demands driven by AI workloads, an analysis of resource utilization on Ascend at OSC, a computing cluster primarily serving AI workloads, reveals that CPU-GPU utilization peaks at only 20–40%, while memory utilization remains as low as 2–10%. This substantial underutilization underscores a critical inefficiency, given the significant energy consumption required to maintain these systems. These findings emphasize the need for estimation frameworks that integrate with resource provisioning systems to enhance allocation strategies and optimize the utilization of shared clusters, ultimately improving energy efficiency in AI-driven data centers.

3.1 Simple Linux Utility for Resource Management (SLURM)

SLURM (Simple Linux Utility for Resource Management)[9] is an open-source job scheduler widely used in high-performance computing (HPC) clusters, including data

[1] https://www.energy.gov/articles/doe-releases-new-report-evaluating-increase-electricity-demand-data-centers?utm$_source = chatgpt.com$

centers and academic supercomputers, for job scheduling and resource management. It offers a scalable and flexible framework for job submission, scheduling, and execution across distributed computing environments. SLURM partitions compute nodes into logical groups and manages job queues based on user-defined policies. It supports batch job scheduling, resource reservations, and fair-share scheduling, making it a preferred choice for supercomputing centers, research institutions, and enterprise HPC environments. Its plugin-based architecture allows customization for various scheduling strategies and integration with service management tools like TAPIS [18]. With a lightweight and fault-tolerant design, SLURM ensures minimal overhead, making it ideal for large-scale scientific simulations, deep learning workloads, and complex parallel computing applications.

SLURM is composed of three primary components: the slurmctld (controller daemon), responsible for job scheduling, resource allocation, and queue management; the slurmd (compute node daemon), which runs on worker nodes to execute tasks and monitor resource usage; and the slurmdbd (database daemon), which integrates with an external database (e.g., MariaDB/MySQL) to store job accounting and usage data for historical tracking and reporting. Users interact with SLURM through command-line utilities such as sbatch (for submitting batch jobs), srun (for running interactive jobs), scancel (for canceling jobs), and squeue (for checking job status). End users specify job requirements in SLURM job scripts, defining parameters like CPUs, GPUs, memory, and execution time, while administrators manage partitions (logical groupings of compute nodes) and configure scheduling policies such as fair-share scheduling, preemption, and priority-based job execution. SLURM supports

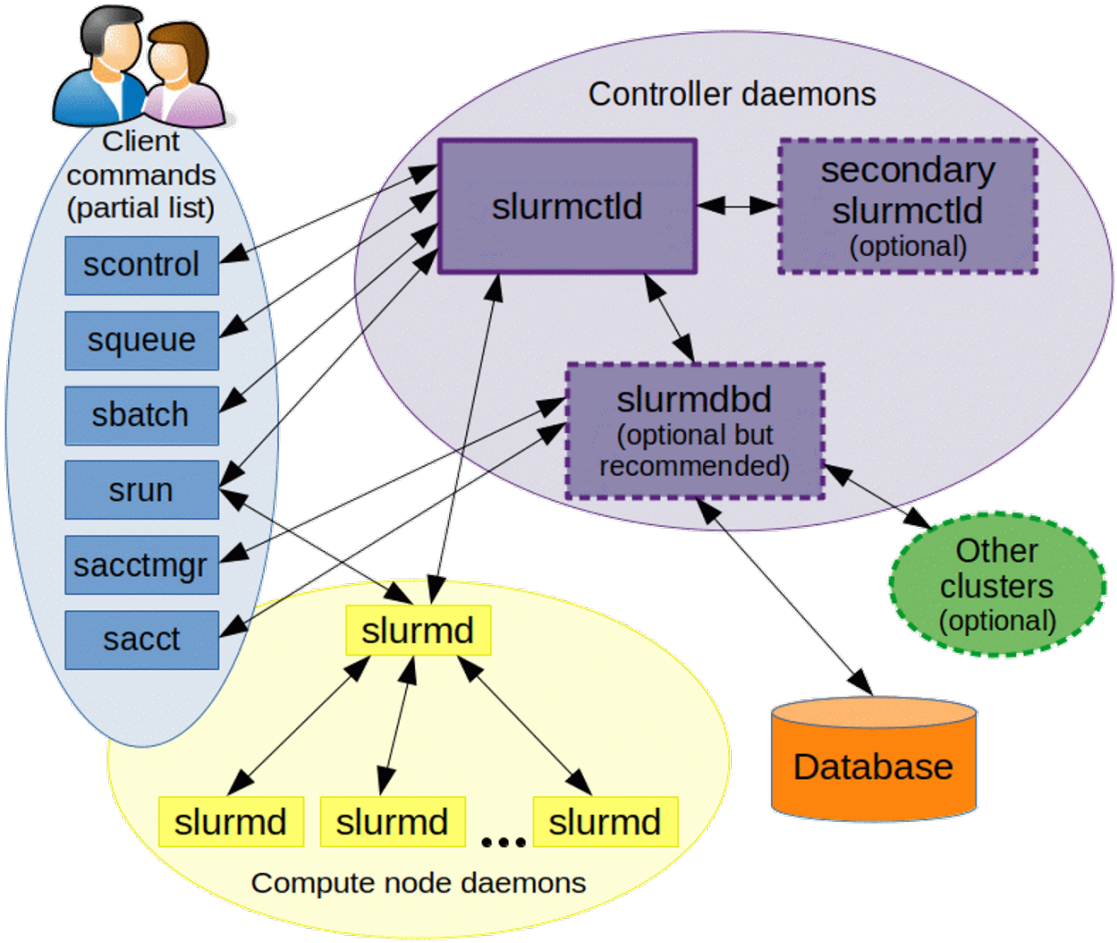


Figure 3.1: SLURM (Simple Linux Utility for Resource Management) Components

multiple scheduling algorithms, including first-come-first-serve (FCFS) and backfilling, ensuring efficient job placement based on resource availability. Its plugin-based architecture allows for seamless integration of custom scheduling policies and machine learning-driven resource prediction, making SLURM a key foundation for developing AI-driven schedulers and resource estimators that leverage its architecture for intelligent workload management

3.2 Existing Resource Estimators

Efficient resource provisioning can be achieved by optimizing several factors, including selecting the best execution endpoint based on resource requirements, estimating near-optimal allocations for successful workload execution, choosing scheduling algorithms that maximize overall utilization, reducing wait times, and improving energy efficiency. Figure 3.2 provides an overview of different estimation approaches that contribute to efficient, estimation-based resource allocation. It highlights the sources of training data and the types of estimators, such as black-box or white-box models and offline or online profiling methods. Most research focuses on estimating either wait time or execution time, as job staging time is theoretically constant and significantly shorter compared to wait times and execution times, particularly for computationally intensive workloads.

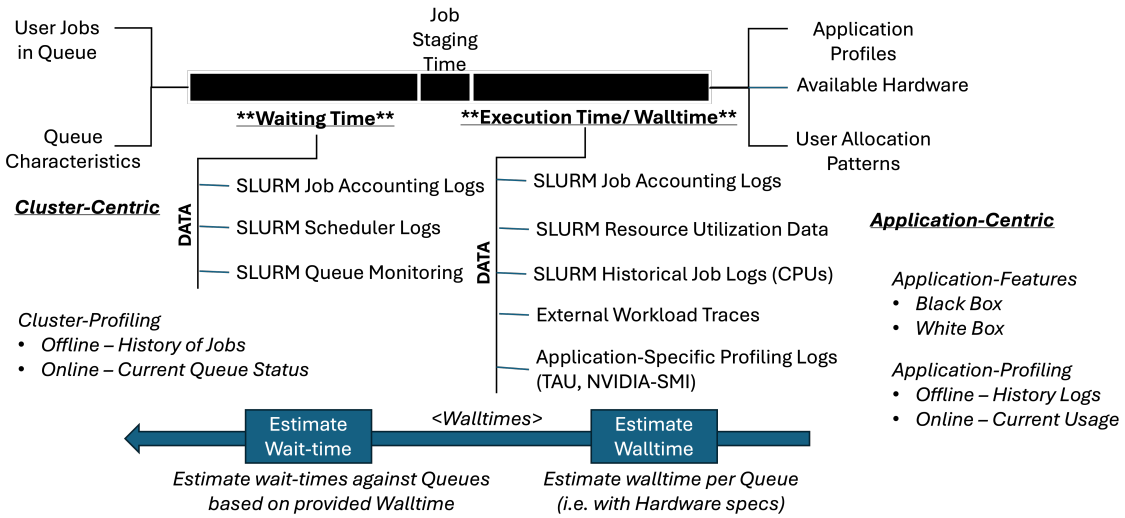


Figure 3.2: An overview of different estimation approaches that contribute to efficient, estimation-based resource allocation

Some research focuses on streamlining wait time estimation [14, 1], which heavily depends on SLURM logs and is specific to the cyberinfrastructure. Estimating wall times and execution times can also be approached from a user-centric perspective, particularly when researchers rely on a specific set of tools (e.g., Trimmomatic, SOAPdenovo) throughout their studies. These tools are frequently used in targeted domains such as genome sequencing. Alternatively, other approaches [7, 8] take an application-centric view, focusing on understanding and estimating the resource demands of deep neural network workloads.

A range of online scheduling algorithms exist, some of which are cyberinfrastructure-agnostic [20], while others are custom-tailored to specific CI environments to estimate wait times. In contrast, runtime estimators can be offline, online, or hybrid. Offline profilers rely on historical execution data to make predictions for future runs or emulate workflows on target environments before making estimations based on these emulations. Online estimators, on the other hand, start by executing a job with an initial configuration and refine predictions based on real-time performance, making them more suitable for pay-by-use models rather than traditional batch scheduling.

Hybrid approaches, such as predicting soft wall times [3], aim to refine runtime estimations dynamically. These methods begin by predicting an initial runtime that is shorter than the user-provided estimate, based on historical allocation utilization. The scheduler treats this as the initial allocation wall time and doubles it incrementally until the job either completes execution or reaches the user-provided runtime. Most resource estimators rely on SLURM log data, including user ID, queue, job submission time, job start time, total execution time, and other scheduling details, rather than actual application execution traces. Similarly, reinforcement learning algorithms

are being explored for offline pre-training and online fine-tuning of scheduling strategies.

While some models use real job traces from supercomputing centers [19, 12], others rely on simulated data [2, 13, 4] due to data center policies restricting the sharing of job traces to protect user privacy and maintain data security. Additionally, the lack of diverse execution configurations in public datasets remains a challenge, as users often either use default configurations or repeatedly submit jobs with the same workable allocations.

Certain models treat runtime estimation as a classification problem [17, 11], rather than the traditional regression-based approach. Although many schedulers share a common feature space—including user ID, nodes, CPUs, GPUs requested, job submission time, job start and end time, and other SLURM job metadata—datasets are often not reusable across studies due to differences in hardware configurations and user workloads. Furthermore, some research papers do not publicly release the real SLURM workloads used to build their schedulers, citing data privacy concerns related to preserving user-HPC interactions.

- Lack of Public Datasets – SLURM logs are often private, and queuing policies vary significantly across supercomputing centers, making it difficult to develop generalizable scheduling and resource estimation models.

- Stale Training Data – Cyberinfrastructure undergoes frequent changes, including hardware upgrades, middleware updates, and software modifications. Additionally, user workloads evolve rapidly, particularly in AI research, where deep neural network (DNN) architectures continuously adapt to newer and more efficient state-of-the-art (SOTA) models, either for improved accuracy or reduced

model size. Existing traces, such as the MIT Data Center dataset [6], may no longer align with current workloads, limiting their applicability in modern research.

In contrast to scheduling-based predictors that rely on SLURM logs (whether real or synthetic), this thesis focuses on application-centric resource prediction. The primary objective is to estimate resource requirements based on the application itself, its parameters, and the available execution endpoints, rather than depending solely on historical SLURM logs. To reduce reliance on stale training data, this approach involves continuous profiling of applications against the current node configurations available in cyberinfrastructure (CI) systems. This ensures that estimations remain accurate and adaptable to evolving workloads and system architectures.

Chapter 4 presents various software stack solutions developed in this research to address the problems identified in Chapter 2 and the limitations of existing approaches. While the proposed method focuses on generating application-centric training data through application execution profiling on targeted hardware, careful consideration is required to ensure that this process remains cost-efficient. Since emulating execution runs consumes valuable CI resources and adds to resource provisioning queues, an optimized approach is necessary. Chapter 5 explores a dual profiling methodology—"Scaled-Down" (SD) and "Full-Scale" (FS)—which balances profiling configurations with a human-in-the-loop feedback mechanism to generate training data while minimizing computational overhead efficiently. We also explore the development of a "Smart-Scheduling" framework inspired by SLURM, proposing the creation of a pre-SLURM CI database to enhance resource estimation and job execution. This database would serve multiple functions, including validating resource

estimations against available queuing configurations, fetching hardware configurations to provide real-time input to inference estimators, and facilitating job execution. Additionally, we propose a job activation daemon that interacts with the database and resource estimators to generate an execution plan (batch submission), submit jobs, and track their progress until completion.

Chapter 4: A Software Framework for Machine Learning-Based Solutions for HPC Optimization

In this section, we present two approaches for delivering AI-driven solutions as a stack of applications and software aimed at addressing the key research questions and current challenges outlined in Chapters 2.3 and 3.

- **Solution1:** We propose the **HPC Application Resource Estimator (HARP)** — a framework designed to generate job-specific profiling data for training off-the-shelf and custom resource (runtime) prediction models using custom loss functions. HARP dynamically selects the optimal model per application at a given instance for inference.

- **Solution2:** We propose the **Smart Scheduler Framework**, which leverages a the estimation models developed in HARP and a **CI database** to create and execute jobs using a **job monitoring daemon**.

Figure 4.1 illustrates the key components developed to address the research questions outlined in Chapter 2. It highlights Solution 1 (HARP)—an AI-driven framework for workload profiling, training data curation, and building and selecting application-specific runtime estimators. Additionally, it presents Solution 2 (Smart Scheduler),

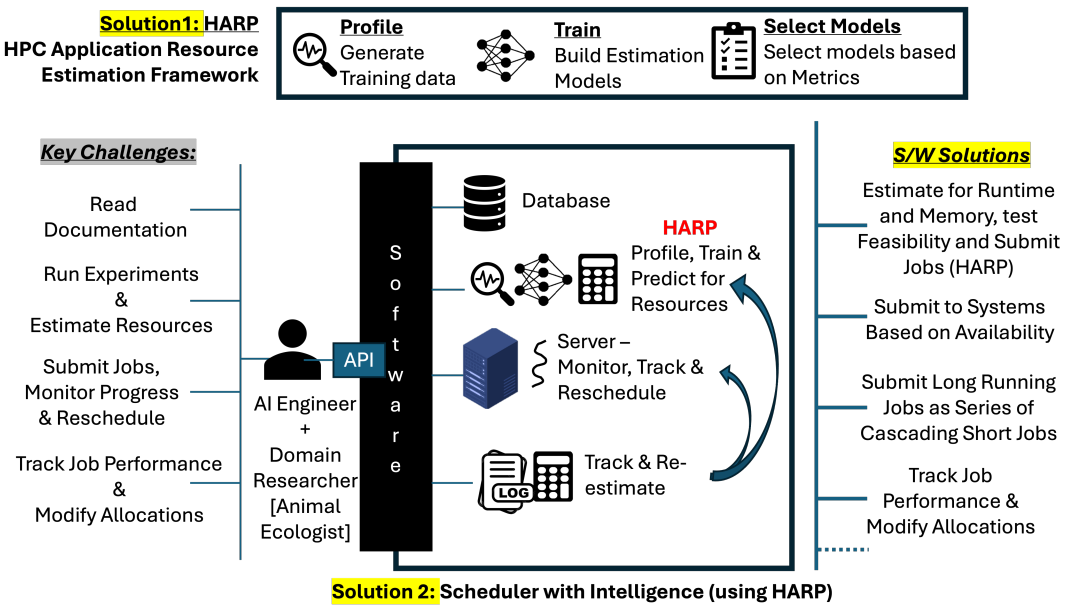


Figure 4.1: Software-Driven Solutions for HPC Resource Optimization.

which focuses on job scheduling, monitoring, and automation to enhance HPC resource management.

Modular Software Component Perspective The above solutions are designed as a stack of modular components, enabling reuse across various end-to-end implementations.

1. **Cyberinfrastructure Database** – A structured database, derived from **CI documentation and hardware configurations**, is utilized in HARP to retrieve system configurations and in the Scheduler to plan execution. See Section 4.3 for further details.

2. **Generating Training Data** – **Offline profiling of workloads** to create datasets for training resource estimators. Refer to Chapter 5 for details on training data generation and its characteristics. The data generation pipeline can be utilized for producing training data for HARP, obtaining inference data in the Scheduler, or in future work where the Intelligence Plane emulates workflows in new environments to adapt predictions to different systems or configurations.

3. **Building Regression Models** – A **pipeline for training resource estimation models**, which can either **auto-schedule jobs** or **validate user-provided resource requests**. Refer to Chapter 6 for an understanding of the loss functions and metrics used in training resource (walltime) estimators, as well as the process of selecting an estimation model to infer runtimes for a targeted application.

4. **Intelligent Scheduler (Intelligence Plane)** – A framework that **automates job submission, monitoring, tracking, and rescheduling**, ensuring jobs run **to completion** efficiently. Refer to Section 4.2 for an overview of the architecture, and to Sections 7.2.1 and 7.2.2 in Chapter 7 for insights into the framework's operation.

These solutions aim to **reduce inefficiencies, improve resource utilization, and automate job management** in HPC environments.

4.1 HARP - The HPC Application Resource Estimation Framework

In this section, we describe the three components - "generate", "build" and "predict" - that form the heart of HARP.

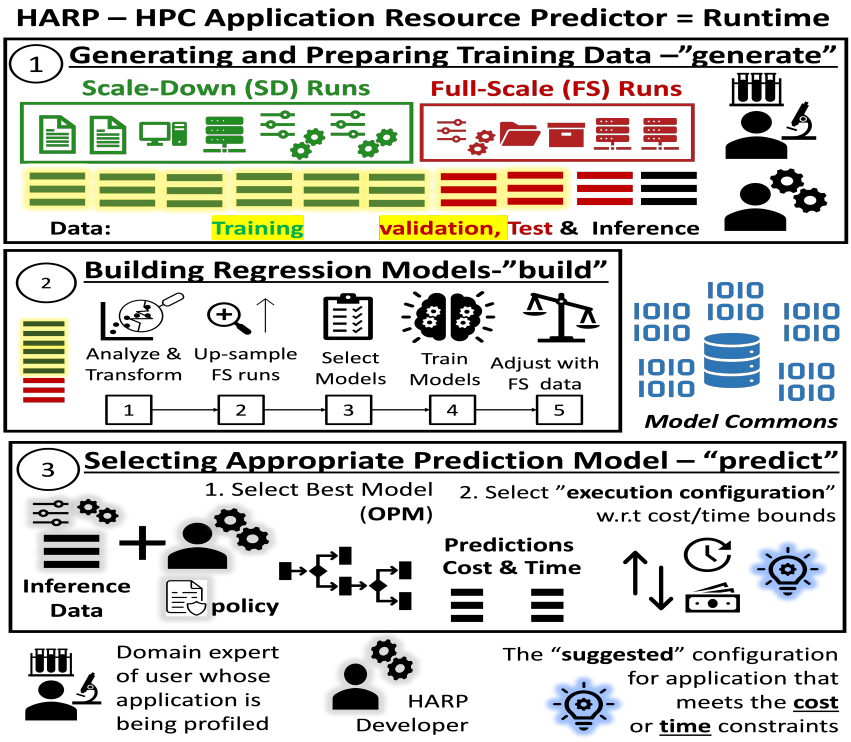


Figure 4.2: Visualizing the Automated Data Collection Pipeline and Model Generation

the Automated Data Collection Pipeline built using Cheetah, TAU, and model generation

1. **The Human-in-the-Loop "Generate" Component:** Utilizing the CODAR Framework [2], this component generates training data by executing applications under scaled-down (SD) and full-scale (FS) configurations. The HARP user defines application-specific hyperparameters and profiling endpoints to generate training data tailored to these configurations.

   The SD configurations are designed to capture a diverse range of tunable workload characteristics, such as different input types and batch sizes for neural

[2]https://github.com/CODARcode/cheetah

networks, and serve as primary training data. In contrast, FS configurations represent actual workload characteristics, such as the expected batch size recommended by the research community, and are primarily used to validate or test "seen/unseen feature" estimations.

As FS workloads become available from user history, they can be incrementally added to the training data, improving model adaptability to real-world workloads. SD executions run applications at a smaller scale, producing lighter and faster workloads, allowing efficient testing across a broad range of applications and allocation configurations. FS runs, on the other hand, scale this information to larger input sizes and hardware configurations, ensuring the model generalizes effectively.

Chapter 5 provides an in-depth understanding of application-specific training data generation with user feedback in the loop, along with several application-centric examples illustrating its implementation.

2. **The Automated "Build" Component:** This component processes the generated training data by capturing all relevant features that influence walltime estimation. These features can include unknown contributors (e.g., memory affecting core allocation per node, which in turn impacts execution time) and presumed features (e.g., 'omp-num-threads' and cores allocated per node), which may introduce redundancy.

   The build component pre-processes training data by removing redundant features and combining correlated features to reduce dimensionality and enhance

predictive model accuracy. Different regression models—Multi-Layer Perceptron (MLP), Decision Tree Regressor (DTR), and Simple Neural Network Regressor (NNR)—are trained using pre-processed data with various combinations of training datasets (SD, SD+n%FS) and stored for inference.

The models are continually trained with incremental FS sample generation (%FS), simulating human learning. To address SD-FS imbalances, FS data is upsampled, and predictions are scaled against the FS validation dataset using an adjustment factor.

3. **The Policy-Driven "Predict" Component:** The final component of the framework predicts execution time for inference (test data). The best-performing model is selected using the Mean Absolute Error (MAE) metric or a custom policy that balances under-prediction and over-estimation, incorporating multiple evaluation metrics such as under-prediction percentage (UPP) and over-estimation mean absolute percentage error (OE_MAPE).

   A model performs optimally when it minimizes UPP while reducing OE_MAPE. Based on this policy, the framework selects the Optimal Predictive Model (OPM) and estimates walltime across all available configurations (hardware and application). The final allocation configuration is then determined based on time and cost constraints, ensuring optimal resource usage.

Chapter 5 outlines all data pre-processing techniques, while Chapter 6 details model training for runtime estimations, enabling the dynamic selection of the optimal estimation model based on the characteristics of the application (i.e., the generated training data).

4.2 iScheduler - The Smart Scheduler Framework

With a diverse range of estimation models and techniques available, as outlined in Chapter 3, it is essential to develop a framework that enables users to interact with these models effectively. However, accessing and utilizing these models remains a challenge for end-users who request resources and schedule jobs, primarily due to the lack of reproducible code or ready-to-use components. To our knowledge, no existing framework seamlessly integrates these models to automate end-to-end job orchestration from development to execution. To address this gap, we propose a smart-scheduler framework, **iScheduler**, designed to orchestrate job execution by leveraging AI-driven resource allocation models. This framework facilitates job monitoring and scheduling through helper functions and API calls, enabling seamless interaction with HPC centers and improving the efficiency of resource provisioning.

The iScheduler Framework (as shown in Figure 4.3) has the following components to orchestrate their jobs:

- Provides an end-user helper tool, a iScheduler Helper Python module, to interact with cloud-hosted estimators for fetching resource requirements and orchestrating job submissions.
- Utilizes a CI database (DB) to store system information and policies, aiding predictions and decision-making before job submission.[3].
- Returns list of feasible predictions (walltime), execution status, and cost feasibility against available systems to end users, enabling them to either submit jobs manually or delegate them (via Intelligence Plane).

[3]We used the CI documentation and SLURM commands to create the DB

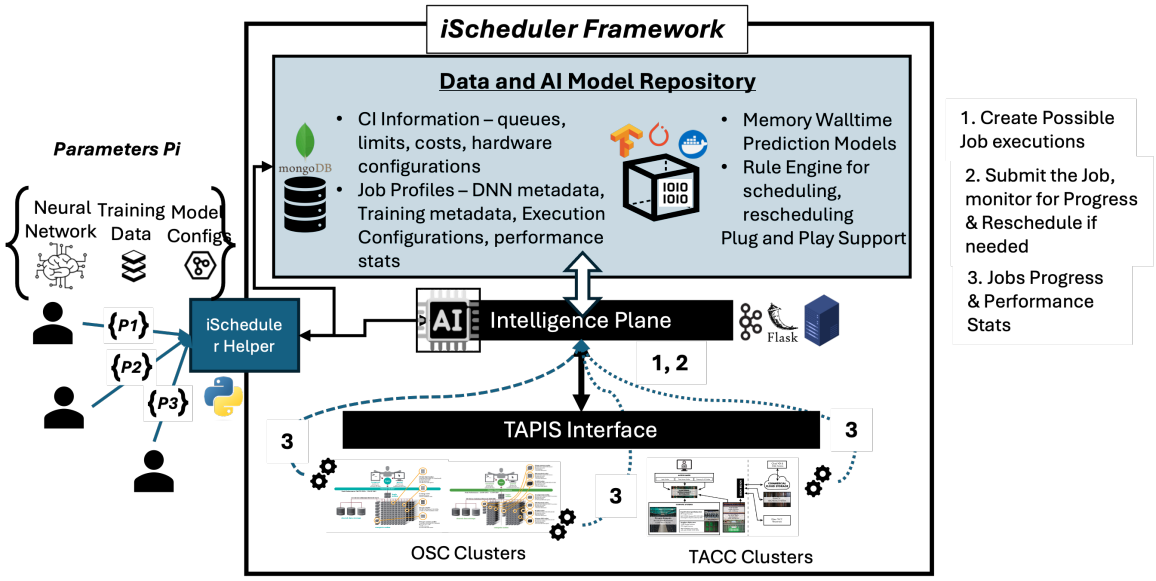


Figure 4.3: Overview of the iScheduler Framework. The framework integrates an AI-powered Intelligence Plane for resource provisioning and scheduling. It leverages a data and AI model repository containing job profiles, execution configurations, and walltime prediction models. The iScheduler Helper, implemented in Python, interacts with TAPIS for execution management.

- Includes an Intelligence Plane (IP) Service for job submission, monitoring, and tracking on behalf of the user (using TAPIS APIs [18])

- Supports plug-and-play upgrades to AI estimators (deployed as container images and executed as TAPIS apps or deployed on ML inference server).

4.3 Cyberinfrastructure (CI) Database: Structure, Implementation, and Applications

This section introduces the Cyberinfrastructure (CI) Database, detailing its development, architecture, and role in HPC batch submission optimization. The CI Database serves as a centralized repository that stores all necessary information to

assist users in creating feasible batch scripts, optimizing resource allocation, and estimating execution costs. As the foundation of intelligent job scheduling, it enables data-driven allocation decisions, execution planning, and feasibility validation.

To enhance accessibility, the CI Database Interface integrates billing formulas and resource allocation policies, ensuring accurate cost estimations. It applies costing models based on supercomputing center policies:

- OSC (Ohio Supercomputer Center): Uses Resource Units (RUs), billing based on CPU and GPU allocations, allowing node sharing.
- TACC (Texas Advanced Computing Center): Uses Service Units (SUs), billing per entire node allocation, as node sharing is not permitted.

4.3.1 Sources of Information for Building the CI Database

The CI database is primarily constructed by scraping supercomputing center cluster webpages, such as OSC - Pitzer [4] and TACC - Frontera [5]. However, the information available on these pages is often limited—for instance, they may not list all available system queues. To address this, we utilize a system development tool [6] to extract detailed system information, including all available queues, CI limitations, and other relevant factors. This tool is executed from cluster login nodes to retrieve node configurations, while additional details, such as client billing costs, are fetched directly from the respective supercomputing center websites.

[4]https://www.osc.edu/resources/technical_support/supercomputers/pitzer

[5]https://docs.tacc.utexas.edu/hpc/frontera/

[6]https://github.com/ICICLE-ai/sysdef_tool

Supercomputing Center Websites

These sources provide costing policies per queue, including details on resource units (RUs/SUs), billing policies, and hardware configurations. Example costing from TACC: A Lonestar6 A100 node equipped with three GPUs incurs a cost of 4 Service Units (SUs) per node-hour, while a Lonestar6 H100 node with two GPUs costs 6 SUs per node-hour. The minimum billing period for both configurations is 15 minutes (0.25 hours). For a 48-hour full allocation, an A100 node would require 192 SUs (4×48), whereas an H100 node would cost 288 SUs (6×48). If a job is terminated early, before completing the minimum 15-minute billing period, the cost would be 1 SU for an A100 node (4×0.25) and 1.5 SUs for an H100 node (6×0.25). These calculations help estimate resource costs and optimize job scheduling based on computational requirements and system constraints. The CI Database integrates this information to estimate costs dynamically based on predicted walltime and system feasibility.

SLURM Interface

As discussed in Chapter 3, Section 3.1, SLURM provides a range of databases that users can query to retrieve relevant information, such as the number of jobs in a queue, the status of their submitted jobs, and overall job scheduling details within the CI. To keep the CI database up to date, we leverage SLURM command-line utilities to extract key system information, including:

- Available queues, their limitations, and access constraints (stored in TABLE: QUEUE_TO_NODE_CONFIG).

- Node configurations, such as maximum memory, available CPUs, and GPUs (stored in QUEUE_TO_NODE_CONFIG and NODE_CONFIG).

Additionally, SLURM commands are used to fetch current queue states, enabling real-time monitoring of node availability at scheduling time.

4.3.2 CI Database Schema and Functionality

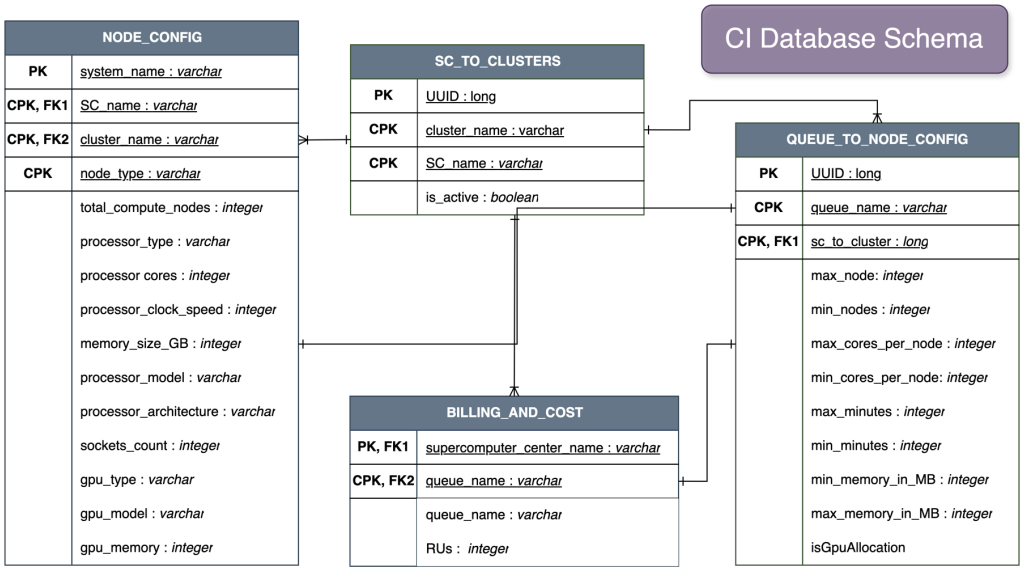


Figure 4.4: CI Database Schema. This schema represents the core components of the Cyberinfrastructure (CI) Database, including system configurations, queue properties, and billing details. The key tables include: *SC_TO_CLUSTERS* maps supercomputing centers to their respective clusters. *QUEUE_TO_NODE_CONFIG* links job queues to their associated node batch limitations. *NODE_CONFIG* stores node-specific hardware configurations. *BILLING_AND_COST* contains information about queue-based billing, including resource unit (RU) costs. This database enables efficient resource allocation, cost estimation, and feasibility validation for HPC job submissions.

Table 4.1: The table shows two cluster allocation policies at two supercomputing centers TACC lonestar6 and OSC Owens. *GPUs come with an additional cost of CPU cores per hour rate.

Queue Name	**Min/Max Nodes per Job**	**Max Job Duration (Hrs)**	**Charge Rate (per core per hour)**
TACC			
development	4	6	1 SU
gpu-a100	16	16	4 SU
gpu-a100-dev	4	2	4 SU
large	65/256	256	1 SU
normal	1/64	96	1 SU
vm-small	1	4	0.143 SU
OSC			
serial	1	168	$0.003
parallel	2/81	96	$0.003
gpuserial	1	168	$0.009*
gpuparallel	2/8	96	$0.009*
hugemem	1	168	$0.004
debug	1/2	1	$0.003

Figure 4.4 illustrates the CI Database schema, which consists of system configurations, queue properties, and billing details. The CI Database Interface enhances this by applying rules on memory per CPU and cost per node-hour, enabling:

- Resource Estimation
  - Supports downstream applications for workload profiling against available system queues and node configurations.
  - Provides CPU, GPU, and memory specifications based on node configurations and queues to feed the resource estimators (inferencing)
- Allocation Feasibility Validation
  - Ensures job requests align with hardware constraints before submission.
  - Helps create execute options againt avaible queues and their feasibilities to enable smart schduling.
- Cost Computation Against Batch Policies
  - Applies cost models based on supercomputing center rules.
  - Optimizes resource scheduling by balancing cost vs. execution efficiency.

Table 4.1 provides a snapshot of the CI configurations database, including selected columns from QUEUE_TO_NODE_CONFIG and BILLING_AND_COST.

The CI Database is a critical component in HPC job optimization, providing a structured framework for resource-aware scheduling, cost estimation, and feasibility analysis. By evaluating resource requirements before job submission, it helps prevent non-feasible allocation requests, reducing the likelihood of jobs being rejected by

SLURM checks or terminated due to errors such as out-of-memory issues. Additionally, the database facilitates the automatic generation of execution plans, including batch scripts, ensuring compliance with system policies while dynamically tuning resource requests based on estimator outputs.

Chapter 5: Generating Training Data to Build Resource Estimators

Estimating execution time and resource requirements is challenging due to several influencing factors. As illustrated in Mind Map Figure 2.2, each application has a unique configuration that impacts memory usage and execution time. Changes in one or more parameters, as shown in Figure 2.4 B, can significantly alter resource requirements. Users typically estimate their resource needs based on previous runs and often reuse the same allocations when re-executing jobs. However, modifying job parameters—such as increasing the number of epochs or adjusting the batch size in DNN training—requires revising resource allocations.

For instance, increasing the batch size demands more memory, potentially requiring a larger GPU node, which is costly and subject to longer queue wait times. Alternatively, offloading the job to a CPU can slow down execution. These patterns form what we refer to as the *history of executions*, which contains records of past job executions across various hardware configurations.

This leads to critical questions:

- How can we access execution history from diverse users and workloads?

- What key features should be extracted from data traces to build an accurate Resource Estimator?

5.1 Challenges for Curating Resource Estimation Training Data

The following section outlines the dataset-related challenges that must be addressed to train a machine learning-based resource estimator effectively.

5.1.1 Availability of Execution History

A well-documented execution history is crucial for training models to learn utilization patterns, including memory requirements, execution times, hardware allocations such as CPU, GPU, and memory, and application-specific parameters like DNN training data, optimization settings, and batch size. However, supercomputing centers often restrict access to workload data due to competition and client confidentiality concerns. While some publicly available datasets, such as the MIT Data Center Challenge Dataset and TPUGraphs, have been developed to analyze and estimate DNN workloads, they come with limitations.

The MIT dataset provides high-level job information, including the DNN model type and allocated resources, but the execution details such as batch size, input data, and optimization settings are stored as low-level time-series logs of approximately 5TB with limited metadata. Extracting meaningful insights from this dataset requires additional models to process execution data for memory and runtime prediction.

Similarly, the TPUGraphs dataset is based on open-source BERT and convolutional models profiled on Google TPUv4. It encodes model architectures using node

features, with each node represented by 140 attributes and adjacency maps for optimization graphs. This dataset, which is about 6GB in size, is designed for Graph Neural Networks (GNNs) and provides a scalable approach for modeling tensor operations. While this allows new architectures to be incorporated into estimators, the dataset is restricted to TPUv4. Since OSC and TACC use GPUs and CPUs, we require a broader dataset that captures resource estimation across multiple architectures. Although models trained on TPUGraphs can predict fit time per training step, they cannot estimate the total resource requirements for SLURM job submissions. Additional models are needed to predict total wall time, making the generation of training datasets for profiling GPU and CPU jobs essential. To address this, we focus on efficiently generating execution traces for target applications, ensuring the process remains quick and cost-effective.

5.1.2 Understanding and Capturing Resource-Specific Features for Machine Learning

Different types of applications have varying resource demands. Compute-intensive workloads, such as deep neural network training, require high computational power and optimized execution pipelines. I/O-intensive workloads, such as processing large text datasets, depend on high disk bandwidth and fast data retrieval mechanisms. Memory-intensive applications, including training large-scale models like BERT, require significant RAM capacity and efficient memory management strategies. Each workload has distinct characteristics that impact execution, including application hyperparameters, hardware attributes like CPU and GPU frequency, number of cores, memory, and bandwidth, as well as concurrency settings such as cores per node and thread count.

Manually engineering these features requires a deep understanding of the tool or application, resource allocation methods, and predictive modeling techniques. Custom feature engineering for each workflow and hardware architecture is often complex, time-consuming, and inefficient. To overcome this challenge, we propose a framework that can automatically extract execution features from historical job logs. By incorporating a human-in-the-loop approach, users can define execution configurations specific to their domain or workload, ensuring that resource estimations remain accurate, adaptive, and scalable across diverse computing environments.

5.2 Human-in-the-Loop Training Data Generation and Processing for Resource Estimation

Generating training data for building estimators involves two key steps:

- Workload Execution and Profiling: Capturing profile dumps by executing workloads on targeted systems.
- Data Processing for Model Training: Processing the profiled data and transforming it into training data for building machine learning models.

5.2.1 Workload Execution and Profiling

To address these challenges, we introduce a human-in-the-loop *generate* component that emulates training data based on user-defined configurations. These configurations are classified into scaled-down (SD) executions, which run at a smaller scale to quickly gather profiling data, and full-scale (FS) executions, which provide validation at production-level configurations. This approach ensures that training datasets are efficiently generated while balancing computational cost and execution speed.

To emulate and profile the identified features for the targeted applications, as described in Chapter **??**, we leverage several third-party tools.

The **Cheetah Experiment Harness and Campaign Management System (Cheetah)** automates the execution of target applications under different system and application configurations. Each job's configuration is specified in a campaign file, which defines execution parameters. Cheetah's runtime module, Savannah, processes metadata from these campaign files to generate execution configurations, referred to as *experiments*, and runs them on the target systems. Every experiment is assigned a unique workspace endpoint, where it stores the execution script, configuration settings, experiment logs, profile traces, and outputs. Additionally, Cheetah includes a post-processing script that collects features from the workspace endpoint after each experiment.

To capture runtime performance metrics, we integrate **Tuning and Analysis Utilities (TAU)** within Cheetah. TAU provides detailed runtime profiling, helping analyze execution behavior. Specifically, we use the `-io` command-line argument to monitor port activity, memory-mapped I/O bandwidths, and maximum bytes read/written per execution. Additionally, we use **Pytools** to extract static profiling information, including processor cores, memory specifications, and other execution environment configurations.

For hardware-specific metrics, we incorporate **NVIDIA-SMI** to collect GPU performance data such as memory utilization, compute load, and power consumption. Similarly, **SLURM** is used to capture CPU-related job metrics, providing insights into CPU time allocation, memory usage, and resource contention.

Finally, **TAPIS** is employed to validate user access to systems listed in the CI Database and to ensure that profiling jobs are executed only on authorized endpoints. This guarantees security and compliance while enabling effective profiling across different computing environments.

Execution Trace Generation Using the CODAR Framework

This component leverages the **CODAR Framework** to generate training data by executing applications under both scaled-down (SD) and full-scale (FS) configurations. The end-user specifies application-specific hyperparameters, while the CI Database retrieves all accessible execution endpoints via TAPIS. Training data is collected from SD runs, which execute the application at a reduced scale, enabling rapid and diverse workload generation across different allocation configurations. Over time, FS data is incrementally incorporated to validate and test the framework against both familiar and novel feature values, ensuring robustness in execution predictions.

FS runs are configured to scale execution traces, allowing the model to generalize across different hardware and input configurations. By combining SD and FS executions, we create an adaptable and efficient data generation process for training ML-based resource estimators.

Training Data Generation for Targeted Applications

This section provides an overview of the methodology used to generate training data for a representative set of workloads for the selected applications outlined in 2.

Application-Centric Features for Training Data Generation Below is a summary of the application-specific features used in generating training data.

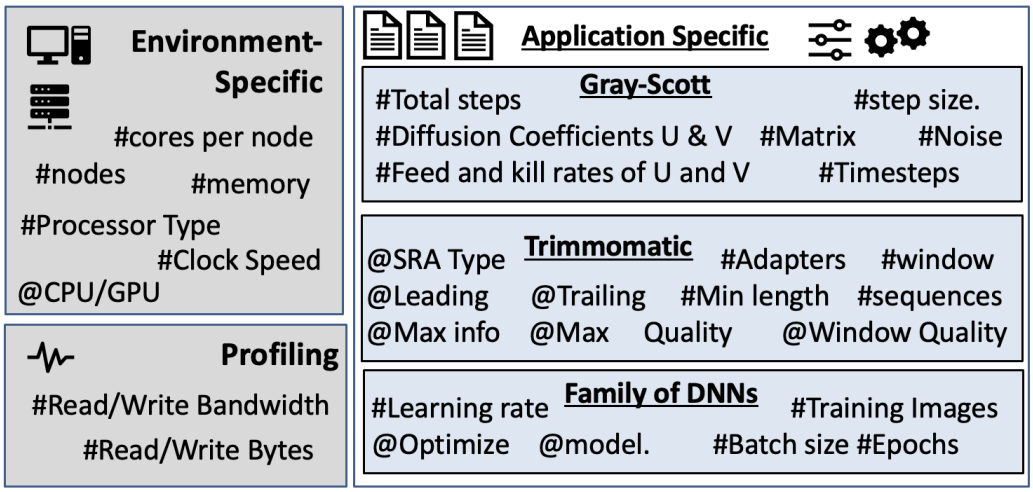


Figure 5.1: Overview of environment-specific, application-specific, and profiling features used in resource estimation. - Categorical Values and #- Numeric values

- For the *Gray-Scott* simulator, experiments were conducted by varying array dimensions, diffusion coefficients of the reacting chemicals, feed/kill rates, and the number of time steps. These simulations were executed across different environments with varying processor allocations. The scaled-down (SD) and full-scale (FS) runs differed in both array dimensions and the number of time steps.

- For *Trimmomatic*, six different genome sequences were used as input short-reads. The trimming quality was influenced by parameters such as leading and trailing quality thresholds, minimum quality per window size, strictness, and minimum sequence length. Higher-quality genome sequences required less preprocessing time compared to lower-quality ones to achieve the same trim configurations. The SD and FS runs differed in genome sequence size, the number of processing threads, and execution environment configurations.

- For the *family of DNNs*, three convolutional neural network (CNN) models were trained using different numbers of training samples and batch sizes. Experiments were conducted on both CPU and GPU nodes with varying core allocations, and execution times were measured based on the average epoch duration. The SD and FS runs differed in training sample count, batch size, number of epochs, and core allocation. FS runs were obtained by training the models on the entire dataset with larger batch sizes, utilizing all available cores of a single-node CPU or GPU HPC allocation.

Figure 5.1 shows an overview of environment-specific, application-specific, and profiling features used in resource estimation. The environment-specific section captures hardware details such as cores per node and processor type. The application-specific section includes domain-dependent parameters for different workloads (Gray-Scott, Trimmomatic, and Family of DNNs). The profiling section captures read/write bandwidth and byte-level I/O operations.

Understanding the Characteristics of Scaled-Down and Full-Scale Data

Executing resource-intensive applications on shared, production research cyberinfrastructures (CI) to generate large amounts of training data is expensive in terms of resources, cost, and time, especially when the application context keeps changing, necessitating the regeneration of training data. To reduce the cost of data generation, we studied the feasibility of generating training samples on commodity environments like workstations or low-contention resources such as single-node allocations. To minimize execution time, we limited input configurations and application features that influence execution time. For example, in DNN training, we reduced the number of

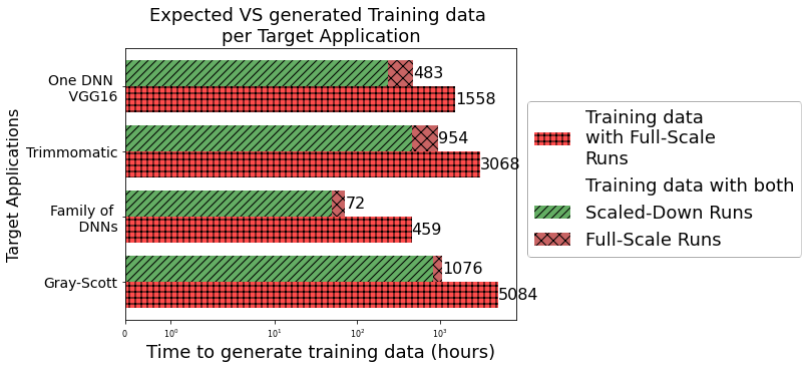


Figure 5.2: Distribution of training data generated using full-scale runs (FS) and scaled-down runs (SD).

images in the training data, creating what we term "scaled-down" (SD) runs. Additionally, we executed the target applications at full-scale (FS) on each new CI for a few iterations to capture CI-specific characteristics in training samples. We profiled the targeted application across a pre-identified set of configurations for both SD and FS runs. The SD executions covered a broader range of feature distributions and were repeated multiple times to exclude errors, while the FS executions focused on capturing CI-specific behaviors. Graph 5.2 presents the estimated cost of generating the entire training dataset using only FS configurations versus the actual cost of generating a combined dataset of SD and FS runs.

Figure 5.3 illustrates the distribution of FS and SD samples generated across these three workflows.

Table 5.1 presents the configurations and characteristics of scaled-down (SD) and full-scale (FS) runs used for training and testing datasets in different combinations. Some observed characteristics of SD and FS samples based on how training, validation, and test data are curated present several challenges:

Table 5.1: Characteristics of different combinations of "scaled-down" and "full-scale" generated samples when used as training and test sets.

	Train & Test	
	FS & FS	**SD & SD**
No pre-processing - "Raw"Generated Training Datasets	-limited training data -multicollinearity exists - limited distribution of samples over feature space	-no feature standardization -multicollinearity exists
Features as Principle Components	-limited training data - limited distribution of samples over feature space	**IDEAL**[b] **Training and Test Data**
Principle Components + Training Data Outliers	-limited training data -outliers exist - limited distribution of samples over feature space	-outliers exist
	SD & FS	**SD+n%FS & FS**[a]
No pre-processing - "Raw"Generated Training Datasets	-unseen data -no feature standardization - multicollinearity exists -training and test have different distribution	-still has unseen data -no feature standardization -multicollinearity exists -imbalanced SD and FS training -training and test have different distribution
Features as Principle Components	-unseen data -training and test have different distribution	-still has unseen data -imbalanced SD and FS training -training and test data have different distribution
Principle Components + Training Data Outliers	-unseen data -outliers exist -training and test have different distribution	-still has unseen data -outliers exist -imbalanced SD and FS training training and test data have different distribution

[a] This is the training data generated through our pipeline, and we want to train highly accurate regression models on this.

[b] The ideal characteristics for training data.
We process the training data intending to achieve some of their characteristics.
SD and FS samples have different distributions for some environment-specific features and input-features.
SD data represents "huge" training data with good sample space.
FS data has "limited" sample distributions over feature-space.

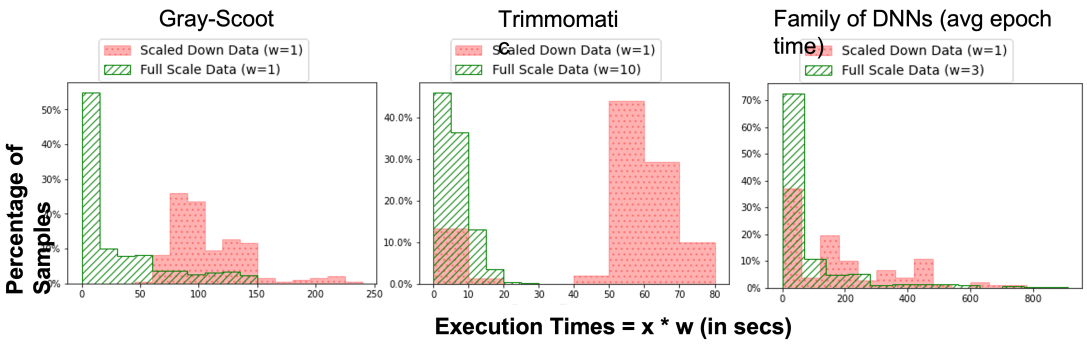


Figure 5.3: Distribution of execution times across different workflows (Gray-Scott, Family of DNNs, and Trimmomatic)

- In an ideal scenario, if the inference/test data follows the same distribution as the training data (IDEAL[b] from Table X), the estimation error should remain minimal, regardless of the model used (see Table X in Chapter Y). However, due to the high cost of generating full-scale (FS) training samples, we cannot produce enough FS data to match the distribution of scaled-down (SD) executions. Using just SD data as training would lead to more scalability errors as the FS test data had "unseen" distribution. Training estimators using only FS samples fails to handle outliers and generalize across diverse workloads effectively.

- Additionally, **multicollinearity** becomes a challenge when two or more independent variables are highly correlated, making it difficult to distinguish their individual impact on the target variable. For example, training sample size and batch size are often strongly correlated, leading to unstable regression coefficients. As the number of identified features increases, the need for more training data grows, creating feature space expansion issues. Without proper

f**eature selection or dimensionality reduction**, this can lead to overfitting and increased computational costs.

- Finally, **scalability** issues arise when transitioning from SD training to FS models, as the limited number of FS samples introduces a severe class imbalance. This imbalance skews model learning, reducing its ability to generalize effectively to unseen workloads. Addressing these challenges requires careful data sampling, feature engineering, and regularization techniques to balance SD and FS training distributions.

5.2.2 Data Processing for Model Training

To address these challenges, we explore **data preprocessing techniques** and the creation of effective train-test-validation splits to optimize the use of emulated training data:

- Training Split: 100% SD + n% FS; Validation Split: n% FS; Test Split: n% FS - Incorporating a small percentage of FS samples into SD data for training enables regression models to scale predictions effectively when executed correctly. The validation dataset helps determine the appropriate scaling factor by minimizing errors, as it shares the same distribution as the test dataset (FS data) for every regression prediction model. However, excessive inclusion of FS samples in a predominantly SD training set may introduce outlier effects, impacting FS run observations.

- To tackle scalability issues and class imbalance, regularization techniques alone may limit the model's ability to learn generalizable scalability patterns from FS data. Sampling methods, such as **upsampling** FS data within the training set,

help align training and test distributions, thereby reducing class imbalance and enhancing model robustness.

- To address multicollinearity, feature selection techniques remove redundant variables, while dimensionality reduction approaches, such as **Principal Component Analysis** (PCA) or feature extraction, retain essential information while reducing complexity. Regularization techniques, such as Ridge Regression (L2 regularization), further mitigate multicollinearity by penalizing large coefficients, improving model stability.

The pipeline "Building Regression Models" in figure 4.2 illustrates a general machine learning workflow. Steps 1 and 2 cover the pre-processing methods described above, while Steps 3, 4, and 5 are detailed in the model development chapter (Chapter 6).

Chapter 6: Runtime Estimators: Training and Selection of Optimal Models for Inference

In this chapter, we discuss various loss functions and evaluation metrics used to assess model accuracy. We then explore both off-the-shelf and custom regression models for runtime estimation, detailing how metrics guide the selection of the best-performing model for a given application based on its characteristics and the availability of historical data (SD and FS samples).

6.1 Loss Functions and Metrics

As we develop estimation models, which primarily rely on regression techniques, they are typically trained using Mean Squared Error (MSE) as the default loss function. To evaluate and compare the performance of these models, metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), or Root Mean Squared Error (RMSE) are commonly used. Although resource (runtime) estimation models are regression-based, underestimations and overestimations have different consequences. Overestimations primarily lead to longer queue wait times, whereas underestimations can be more detrimental, often causing premature job termination and necessitating re-execution, leading to wasted computational resources. To address this imbalance, conventional loss functions need modifications to penalize underpredictions

more heavily, ensuring better optimization during training. This type of adjustment is known as a biased loss function, and one well-known example is the Huber Loss function.

- Mean Squared Error (MSE):The Mean Squared Error (MSE) measures the average squared difference between actual and predicted values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6.1}$$

  MSE has the advantage of penalizing larger errors more than smaller ones, making it useful for models that require higher sensitivity to outliers. However, the same characteristics make MSE is highly sensitive to outliers due to the squared term, which can disproportionately affect the loss

- Root Mean Squared Error (RMSE) The Root Mean Squared Error (RMSE) is the square root of the MSE, which brings the error back to the original unit of measurement:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{6.2}$$

  RMSE maintains the same unit as the target variable, making it more interpretable and incomparable.

- Mean Absolute Percentage Error (MAPE) The Mean Absolute Percentage Error (MAPE) measures error in percentage terms, making it scale-independent:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{6.3}$$

  MAPE expresses error as a percentage, making it easier to interpret across different datasets and is less sensitive to extreme values than MSE and RMSE;

however, it cannot be used when actual values contain zero and tends to overweight small actual values, potentially skewing results

- Huber Loss: Huber Loss is a combination of MSE and MAE, designed to be robust against outliers by applying a quadratic loss for small errors and a linear loss for large errors:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{for } |a| > \delta \end{cases} \tag{6.4}$$

where: $a = y - \hat{y}$ is the residual (difference between actual and predicted values). δ is a threshold that determines the transition between MSE and MAE. Huber loss is less sensitive to outliers compared to MSE and RMSE, providing a balanced approach between squared and absolute error penalties. However, it requires tuning of the threshold parameter δ, which can be challenging, and is more complex to implement than MSE and RMSE.

- Under-Prediction Percentage in Resource Provisioning Under-prediction percentage measures the frequency at which a resource estimator predicts fewer resources than actually required, potentially leading to job failures due to insufficient allocation. It is defined as:

$$\text{Under-Prediction Percentage} = \frac{\sum(\hat{y}_i < y_i)}{N} \times 100 \tag{6.5}$$

where $\hat{y}_i$ is the predicted resource requirement for job i, y_i is the actual resource usage, and N is the total number of jobs.

Under-prediction percentage plays a crucial role in resource provisioning by ensuring conservative estimation and minimizing job failures caused by insufficient resource allocation. A lower under-prediction percentage helps optimize scheduling efficiency by refining estimations over time, thereby improving overall workload performance. Accurate prediction prevents job resubmissions and unnecessary system crashes, leading to better system reliability. Furthermore, minimizing under-prediction enhances adaptive learning in machine learning-based resource estimation models, ensuring that workloads receive adequate computing resources while balancing efficiency and cost.

- Asymmetric Custom Loss for Runtime Prediction: Developing a specialized loss function [5] that optimizes the model to reduce under-predictions while minimizing the percentage error in overestimation.

$$L_H(x) = \begin{cases} -\gamma_L(2x + \gamma_L), & \text{for } -\infty < x < -\gamma_L \\ x^2, & \text{for } -\gamma_L \leq x < 0 \\ x^2, & \text{for } 0 \leq x < \gamma_R \\ \gamma_R(2x - \gamma_R), & \text{for } \gamma_R \leq x < \infty \end{cases} \tag{6.6}$$

  where $x = y_i - \hat{y}_i$, γ_L and γ_R are tunable constants to reduce the no. of underpredictions and minimize the overestimation error.

Figure 6.1 visualizes different loss functions, highlighting the impact of asymmetric penalties on overestimation and underestimation. The left-plot compares multiple Huber loss functions under various γ configurations, while the right-plot illustrates variations of the asymmetric Huber loss function for different asymmetry parameter values (γ_L and γ_R).

To fine-tune models for reducing underpredictions (reducing reruns) and minimizing overestimation errors (avoiding unnecessary wait times), optimizing the hyperparameters γ_L and γ_R is essential. From the right plot, the ideal loss function follows the "red-dashed line," which corresponds to $\gamma_L = 0.1$ and $\gamma_R = 0.9$.

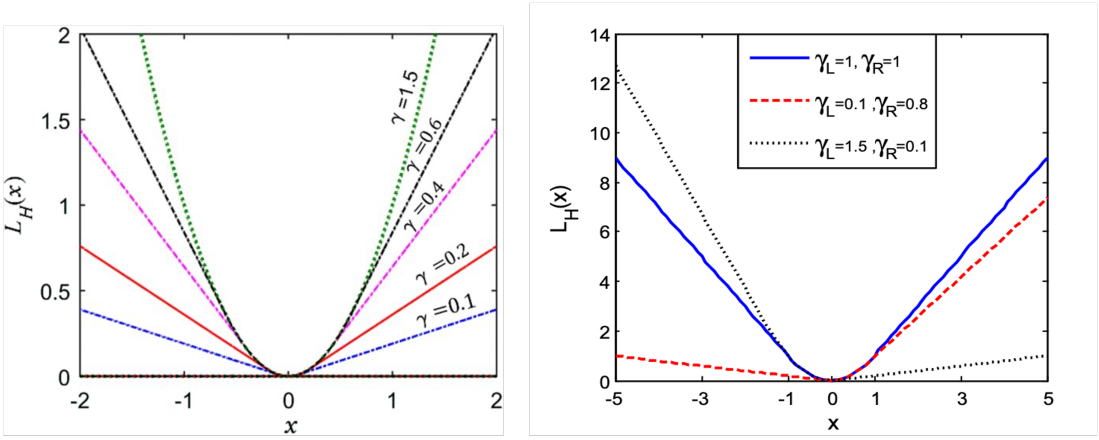


Figure 6.1: Comparison of different Huber-like loss functions for asymmetric error handling. The left plot illustrates the effect of different values of γ on the shape of the loss function, while the right plot presents multiple parameter settings for asymmetric loss variations, controlling how errors are penalized differently for underestimation and overestimation with γ_L and γ_R

6.2 Candidate Regression Models

Given the diverse characteristics of the training data, as shown in Figure 5.1 for each application, a single-model approach is not effective for all cases. For instance, Decision Tree regression models perform best when the training and test sets share the same distribution—both from scaled-down data or full-scale data (see Table **??**). However, a neural network-based or Linear Regression-based approach scales more effectively when training on scaled-down data and inferring on full-scale.

6.2.1 Regression Models and Training Data Characteristics

Decision Tree Regressor

Description: A non-parametric model that splits the data into decision nodes based on feature thresholds. It recursively partitions the dataset into smaller subsets, forming a tree structure.

Ideal Training Data Characteristics:

- Works well with non-linear relationships.
- Handles both categorical and numerical features.
- Performs well with moderate-sized datasets but is prone to overfitting on small datasets.
- Sensitive to noisy data and requires pruning or regularization to prevent overfitting.

Multiple Linear Regression

Description: Multiple Linear Regression (MLR) is an extension of simple linear regression that models the relationship between a dependent variable Y and multiple independent variables $X_1, X_2, ..., X_n$. It assumes a linear relationship between the predictors and the response, expressed as:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon \tag{6.7}$$

Where:

- β_0 is the intercept, $\beta_1, \beta_2, ..., \beta_n$ are the regression coefficients, and ϵ is the error term.

Ideal Training Data Characteristics:

- **Linearity Assumption:** The relationship between independent variables and the dependent variable should be approximately linear.
- **No Multicollinearity:** Independent variables should not be highly correlated with each other. Variance Inflation Factor (VIF) analysis helps detect multicollinearity.
- **Sufficient Data Points:** Works best with a dataset where the number of observations is significantly greater than the number of independent variables to avoid overfitting.

Lasso Regression (L1 Regularization)

Description: A linear regression model with L1 regularization that shrinks some coefficients to zero, effectively performing feature selection.

Ideal Training Data Characteristics:

- Works well when there are many correlated features, as it selects the most relevant ones.
- Suitable for high-dimensional datasets where feature reduction is needed.
- Requires features to be standardized or normalized for better performance.
- Works best when some features are irrelevant or redundant in predicting the output.

Support Vector Regression (SVR)

Description: A kernel-based regression model that finds a function within a margin of tolerance (epsilon) while minimizing prediction errors. Uses kernel functions (linear, RBF, polynomial) to capture complex relationships.

Ideal Training Data Characteristics:

- Effective for small to medium datasets with complex relationships.
- Performs well with high-dimensional data, especially with kernel functions.
- Requires normalized features for optimal performance.
- Sensitive to hyperparameter tuning (e.g., kernel type, epsilon, and regularization parameter).

Each regression model has its own strengths and weaknesses based on the characteristics of the data. Selecting the appropriate model depends on factors such as dataset size, feature complexity, and the presence of noise or collinearity. Table 6.1 provides a side-by-side comparison of these models, outlining their key characteristics for a comprehensive evaluation.

6.3 Model Selection Criteria

Assign Models (A, B) =

$$\begin{cases} BetterModel(TM1, TM2), & UPP(TM1) < UPP(TM2) \\ BetterModel(TM2, TM1), & \text{otherwise} \end{cases} \tag{6.8}$$

Where: TM = Target Model - A binary decision process across all available models.

Table 6.1: The list of regression models and the observed characteristics

Regression Model	Handle Non-linearity?	Needs Huge Training Data?	Sensitive to Outliers?	Handle Categorical Data?	Handle multi-collinearity?	Needs Std.?
SLR	No	No	Yes	No	No	Opt
LAR	No	No	Yes	No	Yes	Opt
DTR	Yes	No	Yes	Yes	Yes	No
SVR	Yes	No	No	No	No	Opt
NNR	Yes	Yes	No	No	Yes	Yes

Table 6.2: This table compares the accuracy of selected regression models based on the type of training and test datasets for the family of DNNs.

	Train & Test	**FS & FS**		**SD & SD**		**SD & FS**	
	RM	**MSE**	**MAPE**	**MSE**	**MAPE**	**MSE**	**MAPE**
No Pre-processing Raw Data**	**NNR**	*513*	*18.76*	*23x*10^5	*31x*10^4	*16x*10^3	*123.35*
	DTR	**35.06**	**2.07**	**18.67**	**7.72**	**3301.52**	**46.85**
	SLR	208.58	10.69	379.00	244.73	1095.06	24.97
	LAS	198.77	10.07	380.16	244.74	1078.22	24.67
Data Processed Data (PCA)**	**NNR**	134.68	8.74	**50.92**	**32.00**	**286.02**	**12.39**
	DTR	**58.14**	**3.22**	**171.90**	**43.09**	3085.47	48.86
	SLR*	208.58	10.69	**409.35**	**252.09**	721.49	18.24
	LAS*	208.45	10.69	**409.35**	**252.05**	721.66	18.25
	SVR	209.77	10.17	**470.64**	**208.80**	1283.43	28.11
PCA + Data with Outliers	**NNR**	*45x*10^5	*373.27*	**94.24**	**31.02**	1444.78	26.47
	DTR	**115.95**	**4.09**	**442.67**	**68.44**	*3620.91*	*52.09*
	SLR*	61x103	197.77	482.75	347.06	***518.62***	***14.98***
	LAS*	60x103	193.31	482.72	347.00	***518.75***	***14.99***
	SVR	249.09	12.30	441.88	217.17	1084.32	24.83

Table \ref{tab:dataconfig} shows the characteristics of these "raw" datasets and the processed training data - PCA Principle Components

** Could not converge while training SVRs on "raw" training data.

* After standardizing the features, SLR and LASSO seem to have similar accuracies, especially with a large number of training samples.

Better Model (A, B) =

$$\begin{cases} A, & (MAE(A) < MAE(B)) \cap (MSE(A) < MSE(B)) \cap (MAPE(A) < MAPE(B)) \\ B, & \text{otherwise} \end{cases} \tag{6.9}$$

(a) Default Criteria (DC)

Better Model (A, B) =

$$\begin{cases} A, & MAE_{\text{change}}(B, A) < w \cdot UPP_{\text{change}}(A, B) \cap \\ & RMSE_{\text{change}}(B, A) < w \cdot UPP_{\text{change}}(A, B) \cap \\ & MAPE_{\text{change}}(B, A) < w \cdot UPP_{\text{change}}(A, B) \\ B, & \text{otherwise} \end{cases} \tag{6.10}$$

(b) Selection Criteria (SC)

Where: w = 1 - Weights to prioritize under-predictions over the general estimation error.

6.4 Model Training and Selection Framework

To ensure optimal performance, our framework must support training a diverse range of regression estimators, including both off-the-shelf models and custom DNN-centric models, and dynamically select the best-performing model at any given time based on the available training data (as shown in Fig. 6.2). The figure illustrates the data availability pattern for training and fine-tuning prediction models. As users execute jobs to test basic configurations or reproduce results, additional execution history (AE) accumulates over time. This enables iterative estimator fine-tuning, enhancing predictions, and adapting estimations to evolving environments.

We adopt a two-step approach for model selection at each retraining step:

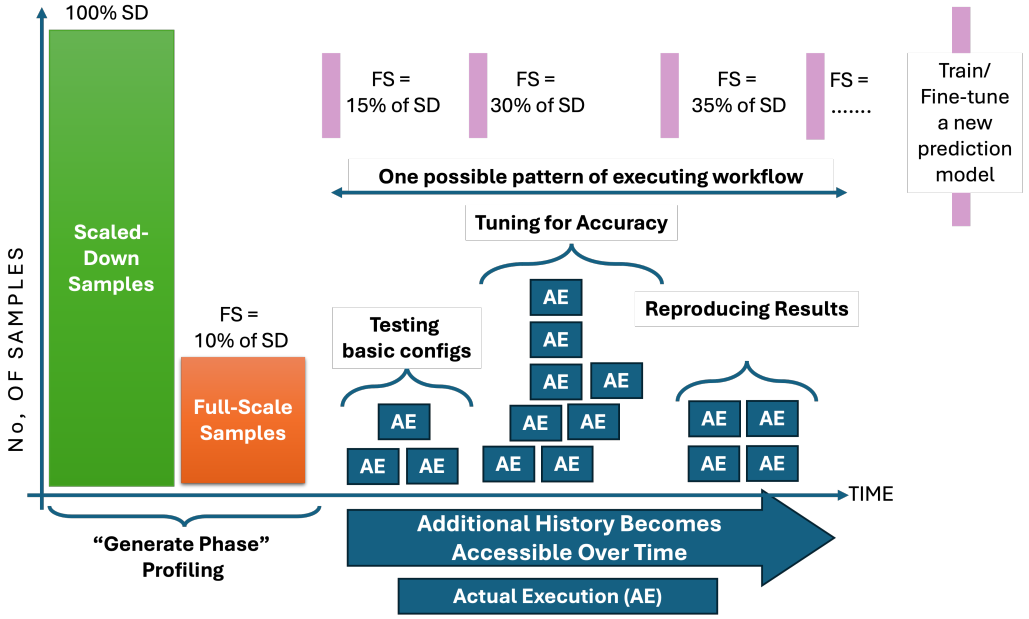


Figure 6.2: Data availability pattern for iterative estimator fine-tuning, leveraging execution history (AE) to enhance predictions and adapt to environmental changes.

1. Train the selected models on the preprocessed training data tailored for a specific target application.

2. Apply the model selection pipeline to identify the best-performing model among the trained models at a given instance for inference.

6.4.1 Training a Pool of Estimation Models

Given a list of available off-the-shelf regression models (see Table 6.1), all models are trained using a custom Huber loss function — an asymmetric custom loss for run-time prediction 6.6. Some models, such as Support Vector Regression (SVR), require data normalization before training, while others, like LASSO and Ridge regression,

converge after applying Principal Component Analysis (PCA) and upsampling. Notably, LASSO and Ridge regression models demonstrate similar accuracy to simple linear regression, making them redundant candidates for regression models (RMs). To optimize performance, we fine-tune the hyperparameters of each model by analyzing the bias-variance tradeoff using cross-validation over the training dataset.

6.4.2 Model Selection with Resource Estimation Criteria

Among all trained models, the best-performing one is selected based on the model selection criteria (Formula 6.10 and 6.9). The Full-Scale (FS) Model, where both training and test samples come from Full-Scale profiles (as shown in Table 6.3), is designated as the Baseline Model. The selected model is then compared against this baseline to assess its performance and robustness.

Our observations indicate that the model selection criteria consistently identify an optimal model for a given application, outperforming the baseline models. We also observe that the selected Model Type changes based on the application type, reinforcing that this pipeline is essential for making the HARP framework more application-agnostic. Additionally, it ensures that the estimators adapt dynamically to the application's characteristics and its available execution history.

Table 6.3: Comparison of the Model Selected by Our Criteria Against the Baseline Model

APP	**Model**	**Training Data**	**Model Type**	**Metrics**		
				UPP	**MSE**	MAPE
Gray-Scott	**Baseline**	FS	SVR	38.30	186.83	12.62
	Selected	SD+ 50%FS	DTR	**32.98**	**120.57**	**4.34**
Trimmo-matic	**Baseline**	FS	SLR	36.84	165.82	3.52
	Selected	SD+ 50%FS	SVR	**7.89**	**1840.32**	**6.58**
Family of DNNs	**Baseline**	FS	DTR	35.00	$10.51 \text{x} 10^4$	176.672
	Selected	SD+ 50%FS	NNR	**15.09**	**5294.590**	**11.800**

Chapter 7: A Use Case: Training Deep Neural Network Models using Smart Scheduler

With software frameworks like HARP, which can profile workloads (such as a family of DNN models or an animal ecology model like MegaDetector v5), build and select specific runtime estimators, and the iScheduler Framework, which invokes these estimators for user workloads, validates executions on the CI database, and monitors job scheduling. This chapter presents two key use cases. First, we demonstrate building scalable estimators using deep neural networks (DNNs), specifically convolutional neural networks (CNNs), which are widely used for image processing and pattern recognition tasks. Then, we showcase a use case in animal ecology, illustrating the interaction of iScheduler with HARP for workload management and execution.

7.1 Building scalable estimators for DNN-based workloads

One of the key bottlenecks in the HARP framework is its dependency on manually generating training datasets. Despite adopting the scaled-down (SD) and full-scale (FS) sampling approach, workloads must still be generated for every new application. For instance, when training a new neural network model for image classification, new profiling data is required. While analyzing estimation models built over a family of convolutional neural networks (CNNs)—referred to as a "Family of DNNs"—we found

Table 7.1: Experiments demonstrate the transferability of regression models within a family of applications.

Source of Training data (Below)	Target Application - TA (Test Data)							
	VGG16				InsV3			
	1LHNN		DTR		1LHNN		DTR	
	MAE MPE	UPP UPAM	MAE MPE	UPP UPAM	MAE MPE	UPP UPAM	MAE MPE	UPP UPAM
VGG16	**1043** **741**	**10.5** **1.4x**	**1272** **369**	**26.31** **1.3x**	3283 1249	0.00 1x	305 252	52.63 2.1x
RN50	3224 1191	5.26 2.1x	253 78	21.05 2.4x	1295 216	15.78 3.8x	1197 465	5.26 4.6x
InsV3	51229 20378	0.00 5.0x	426 179	36.84 3.7x	**1041** **603.08**	**0.00** **1.1x**	**1058** **545.24**	**0.00** **2.4x**
**MIX	**139** **81**	**10.52** **1.5x**	1733 509	10.52 1.8x	524 123	0.00 2.0x	1246 808	5.2 1.6x

** MIX =>usining estining training data with a few profiles from target application to build estimators. Models = {VGG16, InceptionV3, ResNet50}, and (Models - TA) refers to removing the scaled-down (SD) training data for the Target Application (TA) while including only full-scale samples from TA. This setup tests the scalability of estimators across different CNN architectures within the same model family.

empirical evidence that models trained on data generated for certain CNN architectures (e.g., VGG16, ResNet50) can scale to other CNNs (e.g., InceptionV3). Table 7.1 presents the scalability performance of two regression models—Simple Neural Network Regressor (1LHNN) and Decision Tree Regressor (DTR)—within the Family of DNNs. Both models treat the CNN architecture as a single feature (denoted as "@model" in Figure 5.1).

Expanding this study, we assess how well the scalability approach extends to adapting from other neural network architectures, such as a fully connected convolutional neural network (FCNN) for image classification, while evaluating its accuracy and cost implications.

Table 7.2: Comparing Runtime Estimators for Predicting VGG16 Training Time: Evaluating estimators built exclusively from VGG16 emulations versus those incorporating VGG16 data into existing CNN-based emulations.

Prediction Model	**Walltime Error (Actual, Predicted)**			**Training Samples**	**New Data Generation Cost (in $)**
	MAE	**MAPE**	**UPP**		
VGG16	1043	701	11	511	5.15
Fully-CNN	**116**	**56**	**0**	**775**	**1.35**

Table 7.2 presents the results of execution time estimation for a target application—training a VGG16 model on Wiki Images to estimate a person's age. In the first experiment, we build estimators for VGG16 using training data generated from its own emulated executions (both scaled-down (SD) and full-scale (FS) runs). Here, we run HARP end-to-end, generating SD and FS emulations, training models on this data, and building the estimators. In the second experiment, we test whether borrowing training data from existing profiles—specifically from a fully connected convolutional neural network (FCNN)—can enhance estimation accuracy. Instead of generating new scaled-down (SD) runs for VGG16, we only emulate full-scale (FS) runs, then train models on a combined dataset that includes both the newly generated FS data for VGG16 and the previously emulated FCNN data (both SD and FS samples). This approach evaluates whether leveraging existing emulated data from similar models can improve estimation performance while reducing the cost of generating new training data.

By leveraging an existing dataset, we learn execution behaviors from similar models and reduce data generation costs from $5.15 to $1.35. The availability of multiple training emulations for similar models improves prediction accuracy across MAE

(Mean Absolute Error), MAPE (Mean Absolute Percentage Error), and UPP (Under-Prediction Percentage) while minimizing the need for additional application-specific emulations.

7.2 iScheduler Workflow for DNN and MegaDetector Training Job Executions

Figure 4.3 outlines the components of the **iScheduler framework**, while Figure 7.1 illustrates user interaction with the framework through the **iScheduler Helper module**. This module allows users to assess their **job's resource requirements** against available systems, facilitating **job execution orchestration** with the **Intelligence Plane**.

In this section, we walk through two use cases:

1. **Training a ResNet50 model using the scheduled framework**, leveraging previously trained models (as discussed in **Section 7.1**) to provide **feasibility options** and execute the training job.

2. **Using an iScheduler framework notebook endpoint** to **validate feasibility and execute training jobs** for **animal ecology model training with MegaDetector**.

7.2.1 Use Case 1: Training a ResNet50

A user aims to train a ResNet50 vision model with 20k images over 50 epochs, optimizing accuracy by testing different batch sizes. They interact with the iScheduler Helper to assess runtime requirements and batch-scaling feasibility across available CI systems. The workloads are executed as single-node allocations managed through

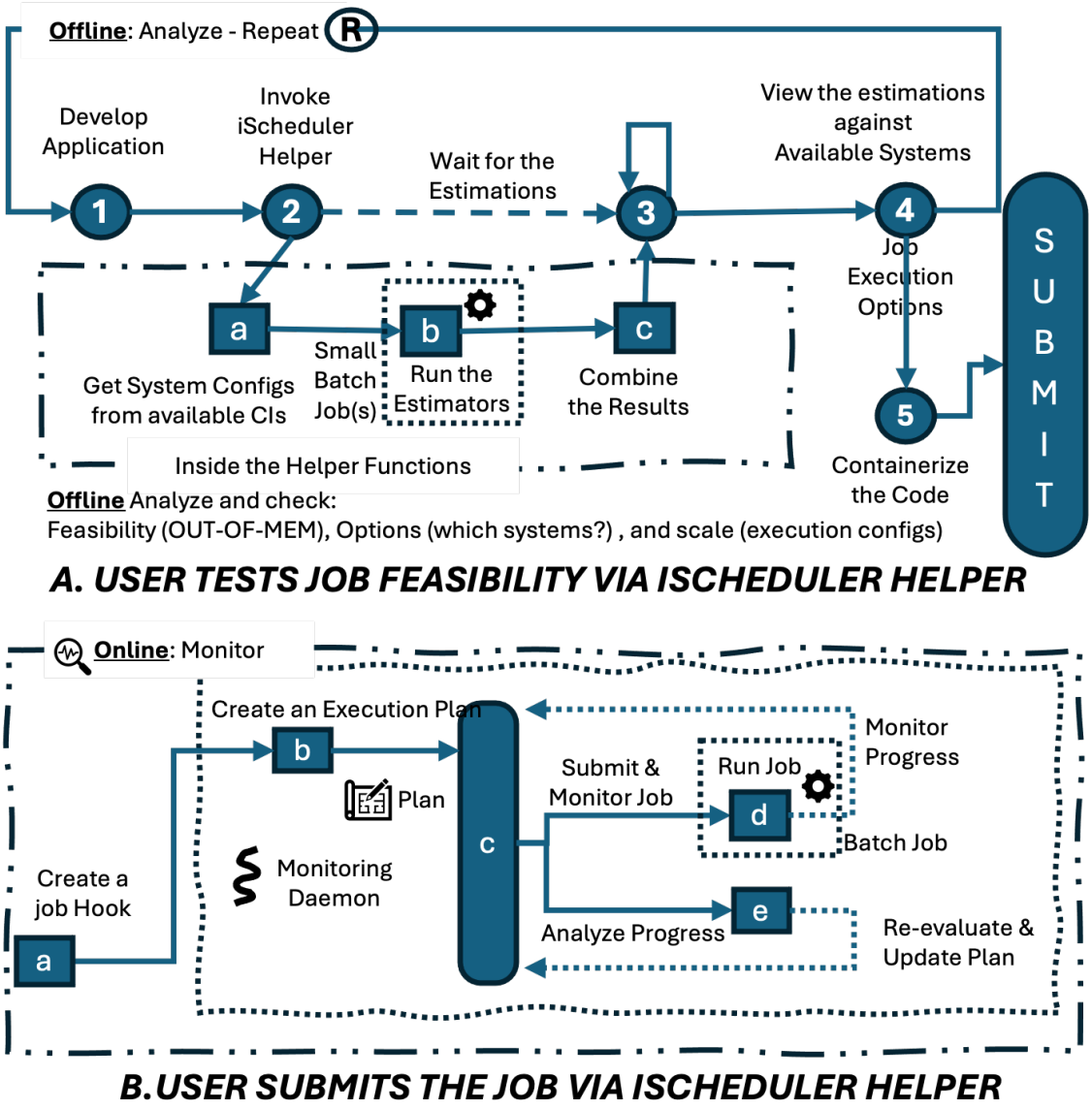


Figure 7.1: Workflow of the iScheduler framework, illustrating both offline analysis and online monitoring phases. The process consists of multiple steps, including workload profiling, resource validation, job submission, and monitoring.

TAPIS, with in-application checkpointing and callbacks ensuring job continuity and monitoring progress.

Workflow: The interaction between the user and the Smart Scheduler follows these steps:

- After configuring the ResNet50 architecture, the user invokes the appropriate iScheduler Helper function, providing model details and target batch sizes (32, 64).
- The Helper Function retrieves node configurations (CPU and GPU node specs for Pitzer Cluster) from the CI database and runs estimators to provide resource predictions.
- The user reviews the estimations and selects an execution plan, either manually submitting the job or delegating it to the Intelligence Plane.
- **Resource Estimation Results:**
  - **Pitzer CPU Node:** Can execute ResNet50 for both batch sizes with estimated execution times of 9.2 hours (batch size 32) and 9.7 hours (batch size 64), with corresponding costs of $0.7728 and $0.7308, respectively.
  - **Pitzer GPU Node (32 GB Memory):** Can handle only batch size 32, with an estimated execution time of 2.4 hours and a cost of $0.4176.
- The user has two options after reviewing estimation results:
  - Manual Submission: Configure a SLURM script to submit the job independently.

  - Delegated Execution: Allow the Intelligence Plane to handle job submission and monitoring.

- The prototype demonstrates two execution feasibilities:
  - Ideal System Configurations: Submitting with the best possible configurations to minimize interruptions.
  - Availability-Based Execution: Adapting to CI queue states, such as number of idle nodes and pending jobs.

- The user containerizes the code, publishes it to a publicly accessible location, and creates a TAPIS Application.

- The Intelligence Plane manages job execution, submission, and progress monitoring via callbacks. Jobs exceeding the maximum walltime are handled as cascading tasks, with reassessments and AI-driven finetuning loops optimizing memory and walltime estimations.

- **Execution Strategy Comparison:**
  - **Ideal Configuration (Pitzer GPU):** Estimated job completion 28 hours (26 hours wait time + 2 hours execution) with a cost of $0.348.
  - **Availability-Based Execution:** Estimated job completion 9 hours (immediate resource allocation) with a cost of $0.756.

Given the GPU node wait time, the current ResNet50 training job could be completed on a CPU node before the GPU allocation becomes available, potentially saving researchers valuable time.

7.2.2 Use Case 2: Training a MegaDetectorV5

The primary objective of this use case is to leverage the iScheduler Framework integrated with TAPIS to help ecologists efficiently fine-tune their models.

Adapting MegaDetector for Small Animal Ecosystem Monitoring

Ecologists study small animal ecosystems by collecting data using the Adapted Hunt Drift Fence Technique (AHDriFT), which enhances detection by guiding small animals into a bucket equipped with a camera trap [**?**]. Conducted by the Ohio Department of Natural Resources, this survey generated a benchmark dataset of 120,000 annotated images, covering 7 order-level and 45 species-level categories, including small mammals, reptiles, and amphibians commonly found in Ohio.

The camera trap edge device deployed in the field consists of a motion sensor, a camera, and a small computing unit (e.g., a Raspberry Pi) to run inference models. When the sensor detects movement, it triggers the camera to capture a burst of images to record the object that activated the motion detection. These images are then analyzed by an inference model, such as MegaDetector [**?**], running on the edge device to classify the object as either an animal or non-animal motion (e.g., leaf movement). Only relevant images are stored, significantly conserving storage space, as approximately 70% of captured images on camera traps are empty frames, which would otherwise quickly fill the device's limited storage.

Challenges in Deploying Pretrained MegaDetector Models

Deploying an off-the-shelf pretrained MegaDetector model on the edge device proved inefficient due to mismatches in training data. Existing MegaDetector models were primarily trained on larger animals, whereas the Ohio Small Animal Survey dataset includes small rodents, insects, and other tiny species. The survey utilizes ground-facing cameras with a consistent background, minimizing noise and false triggers. In contrast, MegaDetector's pretrained models were trained on side-facing camera traps, capturing highly varied environmental backgrounds, leading to greater variation in lighting conditions, occlusions, and motion artifacts.

Due to these substantial differences, researchers need to fine-tune the MegaDetector model on this representative dataset before deployment on the edge device. Fine-tuning ensures the model effectively identifies small animals while filtering out irrelevant detections.

Interactive Training & Scheduling with iScheduler

Researchers can interact with the iScheduler Notebook Interface [**?**] to assess the cost and feasibility of fine-tuning MegaDetector and execute training jobs using the scheduler interface. The iScheduler Notebook or TAPIS can be used to monitor training progress, whether as a single job or a series of sequential jobs scheduled based on resource availability. This dynamic scheduling approach ensures efficient resource allocation for fine-tuning, improving model adaptability for small-animal detection in edge environments.

Figure 7.2 illustrates the steps an ecologist follows while interacting with the **iScheduler Framework**:

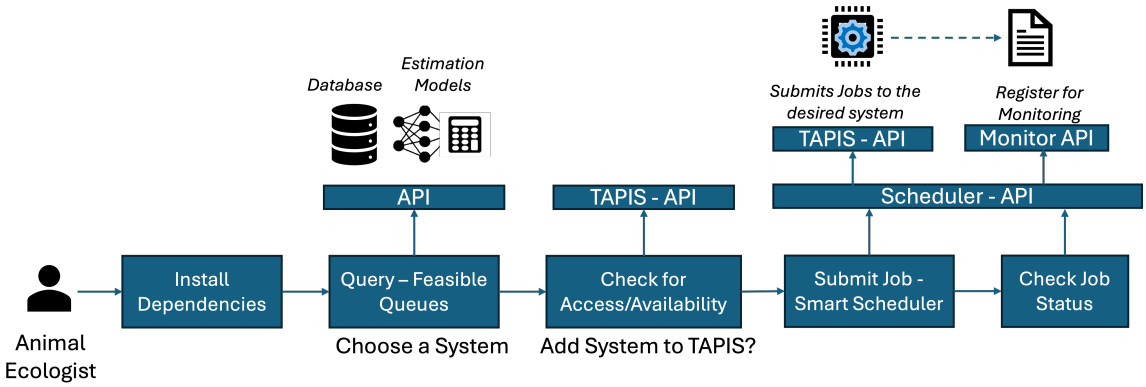


Figure 7.2: Workflow illustrating the steps an ecologist follows when interacting with the iScheduler Framework. The process includes installing dependencies, querying feasible queues, checking access, submitting jobs, and monitoring execution status through TAPIS and Scheduler APIs.

1. **Install Dependencies:** Download and install the necessary components, including the Smart Scheduler and MegaDetector dependencies, from GitHub.

2. **Query Feasible Queues:** Interact with the iScheduler Database Helper functions via the API to determine feasible execution queues for available systems registered in TAPIS (including all OSC and TACC resources). Table 7.3 shows a snapshot into the estion feasibility results.

3. **Check for Access and Availability:** Verify access permissions for the identified systems. If a system offers a better cost-turnaround time, users may request access to TACC or OSC.

4. **Submit Jobs via Smart Scheduler:** Submit the job using the Smart Scheduler API, which handles execution on the selected system.

Table 7.3: Cost estimation for fine-tuning the MegaDetector model based on fine-tuning predictions across different queues.

cluster_name	queue_name	MaxMemory (MB)	Max Minutes	Feasible	Computed _Cost ($)
owens	condo-PYS1124-backfill-serial	120820	360	No	0.000
owens	gpubackfill-parallel	120820	240	Yes	22.320
owens	debug	120820	60	No	0.000
pitzer	condo-osumed-gpu-48core-backfill-serial	371712	240	Yes	22.320
pitzer	CPU-serial	182240	360	Yes	1.080
pitzer	condo-gao-gpu-backfill-serial	371712	60	No	0.000
pitzer	condo-osumed-gpu-48core-backfill-parallel	371712	240	Yes	22.320

5. **Monitor Job Status:** Track job execution through the Scheduler API or directly via TAPIS Monitor API.

Chapter 8: Future Work

The future work is distributed across four key directions:

- **Training White-Box Estimators:** After observing how estimation models built for a family of applications (e.g., CNNs) scale empirically to newer architectures within the family (See Chapter 7 Section 7.1), we extend our approach to "featurizing" the model rather than treating it as a single feature (@model). This enables better scalability and further reduces the dependency on generating even full-scale (FS) data for new models or architectures. This approach relies on high-level optimization (HLO) graphs generated by the XLA compiler in TensorFlow, which are used to create graph-based features such as node attributes and adjacency matrices.

- **Synthesizing High-Level Optimization Graphs for a Given Model:** A key bottleneck in the white-box approach is the requirement to extract model-specific features from HLO graphs, which necessitates running the model on every architecture at least once to generate the HLO. This dependency on OSC or TACC resources for inference sample collection adds computational overhead. To address this, we explore the use of Generative Adversarial Networks (GANs)

to synthetically generate HLO representations instead of extracting them from actual executions, thus mitigating this bottleneck.

- **Reimagining the "Intelligence Plane," an Agentic AI for MLOps:** MLOps (Machine Learning Operations) integrates DevOps principles with machine learning workflows to enable scalable, automated, and reproducible model deployment. The Intelligence Plane (IP) can act as an autonomous AI agent for MLOps, optimizing job scheduling, monitoring resource allocation, and dynamically adapting inference models based on workload changes. By integrating IP as an agent, we aim to build self-optimizing MLOps pipelines capable of automated decision-making and real-time performance tuning.

- **Developing an LLM for a CI MLOps Agent:** We propose integrating large language models (LLMs) into the CI database and exposed APIs, creating a conversational interface for users. This interface can assist researchers by answering queries related to resource estimation, job scheduling, and cost analysis. Additionally, the LLM can explain estimation computations, provide recommendations, and help users navigate the scheduler framework. By bridging natural language processing (NLP) with CI MLOps, this approach enhances user accessibility and improves interaction with complex HPC workflows.

8.1 Training White-Box Estimators

8.1.1 Background

This section explores the relationship between computational graphs, DNN workload execution (e.g., training), and hardware-aware features for GPUs, highlighting the role of Graph Neural Networks (GNNs) in developing new white-box estimators.

Computational Graphs in DNN Frameworks

Machine learning frameworks like TensorFlow and PyTorch represent DNN models as computational graphs, where nodes represent operations and edges define data dependencies. This graph structure enhances efficiency and optimization, especially for training large models. The advent of ML compilers, such as XLA (Accelerated Linear Algebra), has refined this process by converting computational graphs into High-Level Optimizer (HLO) graphs, an intermediate representation that optimizes computational workflows. HLO graphs break down complex DNN operations into smaller, hardware-optimized tasks.

Studies, such as TPUGraphs [16], show that HLO graphs, particularly when deployed on accelerators like TPUs, improve performance by optimizing dependencies and memory usage. This approach informs our project, where HLO-extracted features enrich the HARP framework for more accurate runtime predictions.

Hardware-Aware Prediction Models

Recent research highlights the significance of hardware-specific features in improving resource prediction models. Prior work [22] has explored GPU-based predictors incorporating attributes such as memory bandwidth and core configurations to estimate DNN training performance across various GPU architectures. This hardware-aware strategy has shown substantial accuracy gains by aligning predictions with device capabilities.

Building on these insights, our approach integrates hardware features (e.g., CUDA cores, tensor cores) into the HARP framework. By incorporating these attributes, the

enhanced HARP estimator produces hardware-adaptive runtime predictions, optimizing model performance on different computing infrastructures.

Graph Neural Networks for Resource Prediction

Graph Neural Networks (GNNs) [10] effectively model complex relationships within computational graphs, making them well-suited for resource prediction in DNN workloads. By capturing hierarchical and sequential dependencies, GNNs provide a more structured representation of computational flow than traditional approaches.

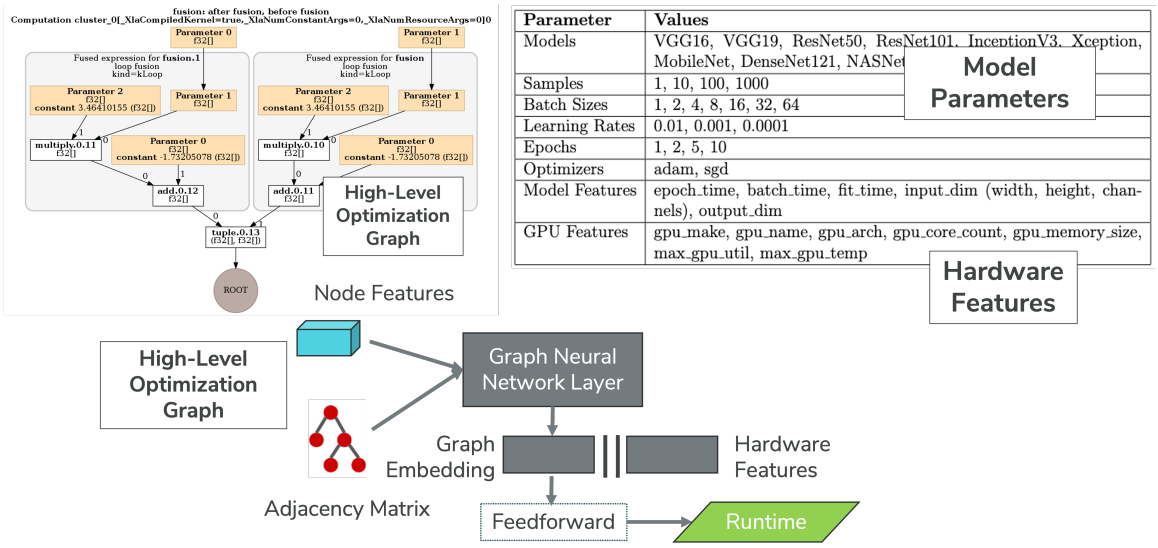
GNN-based models have been successfully applied to TPU-inspired resource prediction frameworks, where they embed node features and adjacency information for improved runtime estimation. Our project leverages GNNs to model DNN computational flows extracted from HLO graphs, enabling the extended HARP framework to generalize across different DNN architectures and adapt to diverse hardware configurations.

8.1.2 Methodology

We configure the HARP framework with an enhanced data capture pipeline, enabling the generation of training data while extracting new features from HLO graphs. Figure X outlines these newly captured features, encompassing both model-specific and hardware-specific attributes, which contribute to building HLO-based white-box estimators.

Dataset Description

Our dataset consists of over 20,000 records, each representing a unique configuration across model architectures, GPUs, and hyperparameters. Data collection was



Parameter	Values
Models	VGG16, VGG19, ResNet50, ResNet101, InceptionV3, Xception, MobileNet, DenseNet121, NASNet
Samples	1, 10, 100, 1000
Batch Sizes	1, 2, 4, 8, 16, 32, 64
Learning Rates	0.01, 0.001, 0.0001
Epochs	1, 2, 5, 10
Optimizers	adam, sgd
Model Features	epoch_time, batch_time, fit_time, input_dim (width, height, channels), output_dim
GPU Features	gpu_make, gpu_name, gpu_arch, gpu_core_count, gpu_memory_size, max_gpu_util, max_gpu_temp

Figure 8.1: Feature extraction using High-Level Optimization (HLO) graphs for training white-box estimators.

conducted across three Ohio Supercomputing Center (OSC) clusters: Owens (Nvidia P100), Pitzer (Nvidia V100), and Ascend (Nvidia A100). The dataset comprises 30 distinct features, covering input-output specifications, GPU-specific attributes, and model performance metrics, as detailed in Table 8.1. Additionally, it includes NPZ files generated using a new feature capture pipeline, which extracts High-Level Optimization (HLO) graphs and features from computational workloads [10].

NPZ File Details Each dataset entry is associated with an NPZ file containing fine-grained computational graph data extracted from HLO graphs. The following key feature groups are included:

- **Node Features**: Descriptive properties for each node in the computational graph:

Feature	Type	Description
Name	Identifier	Unique identifier combining model, optimizer, sample size, input dimensions, and parameters
samples	Integer	Number of samples used for each configuration
input_dim_w, input_dim_h, input_dim_c	Integer	Input dimensions (width, height, channels)
output_dim	Integer	Number of output classes
epochs	Integer	Number of training epochs
batch_size	Integer	Training batch size
learn_rate	Float	Learning rate for optimization
tf_version	Categorical	TensorFlow version for reproducibility
cuda_version	Categorical	CUDA version used by TensorFlow
batch_time	Float	Time taken per batch
epoch_time	Float	Time per epoch
fit_time	Float	Total training time
npz_path	String	Path to associated NPZ file
gpu_make, gpu_name	Categorical	GPU manufacturer and model
gpu_arch	Categorical	GPU architecture (e.g., Ampere for A100)
gpu_cc	Categorical	GPU compute capability
gpu_core_count	Integer	Number of CUDA cores
gpu_sm_count	Integer	Number of streaming multiprocessors
gpu_memory_size	Integer	GPU memory size (GB)
gpu_memory_type	Categorical	Type of GPU memory (e.g., HBM2e)
gpu_memory_bw	Integer	Memory bandwidth (GB/s)
gpu_tensor_core_count	Integer	Number of tensor cores
max_memory_util, avg_memory_util	Float	Maximum and average GPU memory utilization (%)
max_gpu_util, avg_gpu_util	Float	Maximum and average GPU utilization (%)
max_gpu_temp, avg_gpu_temp	Float	Maximum and average GPU temperature ()

Table 8.1: Dataset Features with Descriptions

- **Root Indicator (is_root)**: Identifies the final output node.
- **Tensor Type Encoding**: One-hot encoded representation of tensor data types.
- **Tensor Shape Features**: Includes shape dimensions (up to six), sum, and product.
- **Convolutional Operations**: Window size, stride, padding configurations.
- **Slice Dynamic Slice Features**: Range indicators and slice sizes.

- **Tile Configuration Features**: Defines tiling strategies for optimizing convolutions.
- **Layout Configuration Features**: Specifies memory layout strategies for tensors.

Dataset Generation Data was generated by systematically varying hyperparameters such as sample size, batch size, learning rate, epochs, and optimizers across 17 different models. Figure **??** outlines the range of model and hardware parameters captured within the dataset.

8.2 Reimagining the "Intelligence Plane" an Agentic AI for MLOps:

MLOps Overview

MLOps (Machine Learning Operations) is a set of practices designed to streamline the development, deployment, monitoring, and automation of machine learning (ML) models in production environments. It integrates principles from DevOps, Data Engineering, and ML Lifecycle Management to ensure scalability, reproducibility, and reliability of ML workflows.

Key Components of MLOps This section outlines the core components of MLOps and how ICICLE components support and implement these functionalities.

- **Model Development & Versioning** – Managing code, data, and model versions for reproducibility. (Handled by **Model Commons**)
- **Automated Training & Testing** – Continuous retraining using CI/CD pipelines. (Using **HARP** and **iScheduler**)

- **Model Deployment & Serving** – Deploying models in production using Kubernetes, Docker, or cloud services. (Using **TAPIS**)
- **Monitoring & Performance Tracking** – Observing model drift, latency, and performance metrics. (Using **Kafka** in the **Smart Scheduler Framework**)
- **Feature Engineering & Data Management** – Ensuring high-quality data pipelines. (Customizable by users, enabled by **TAPIS Workbench**)

MLOps facilitates seamless collaboration between data scientists, engineers, and IT teams, ensuring robust and efficient AI model lifecycle management in production.

ICICLE Intelligence Plane Overview

The ICICLE Intelligence Plane is a federated subsystem that governs performance metrics, resource allocation, and AI-driven workflows within the ICICLE ecosystem [15]. It facilitates the construction, deployment, and optimization of intelligent components, including machine learning models, middleware, runtime schedulers, and resource estimators.

Key Components of the Intelligence Plane

- **Smart Scheduler** – Estimates training times, evaluates queue conditions, and schedules jobs for optimal execution.
- **Edge Trigger Mechanism** – Monitors model accuracy and triggers retraining events based on predefined thresholds.
- **HARP Pipeline** – Optimizes resource allocation when estimator performance declines.

- **Job Progress Monitor** – Tracks job execution and provides real-time updates on model training and deployment.
- **Model Cards** – Documents and registers new models (e.g., smaller architectures, improved versions) for reproducibility and transparency.
- **Event Handling Framework** – Detects and processes key retraining and execution events:
  - New data availability.
  - Insufficient images in the observation folder.
  - Job execution status updates.
  - Model accuracy drops below a predefined threshold over N inferences.
  - Health check failures in estimation models.
  - New model registration triggers (e.g., smaller models, new architectures).
- **Rules Database** – Stores and maintains monitoring policies:
  - Tracks applications and their performance metrics (accuracy, energy consumption, etc.).
  - Stores model metadata in **Data Commons** while summarizing statistics in **Component Commons**.
- **Watch Folders** – Continuously monitors new training data (e.g., images from edge devices) to trigger automated processing.
- **Service Orchestration Engine (Daemon Endpoints)** – Manages job scheduling, execution monitoring, and system performance tracking:

  - **Job Monitoring Daemon** – Uses **SLURM** and **Kafka** to monitor job performance and update the **Component Start Database**.
  - **Estimator Health Checker** – Periodically validates the accuracy and efficiency of estimation models.

- **Training Data Generation Module** – Expands training datasets by running phase-one **HARP-return models** in new environments or integrating updated versions (e.g., MDv5, MDv6) into the training pool.

The ICICLE Intelligence Plane facilitates a fully automated, AI-driven scientific workflow, incorporating model retraining, event-driven execution, and real-time performance monitoring to enhance adaptability and efficiency across various research applications. Figure 8.2 illustrates the current implementation of the Intelligence Plane alongside the proposed Intelligence Plane v2.0 architecture.

Intelligence Plane as an Agentic AI for MLOps

Agentic AI refers to AI systems that autonomously perceive, reason, and act to achieve specific goals with minimal human intervention. Unlike traditional AI, which passively processes inputs, Agentic AI actively interacts with its environment, makes adaptive decisions, and refines its behavior over time.

Key Components of Agentic AI and ICICLE's Alignment

- **Perception Module** – Gathers real-time data from sensors, databases, or APIs. (Implemented through **Watch Services, Edge Triggers, Job Monitors, and CKN** [21])

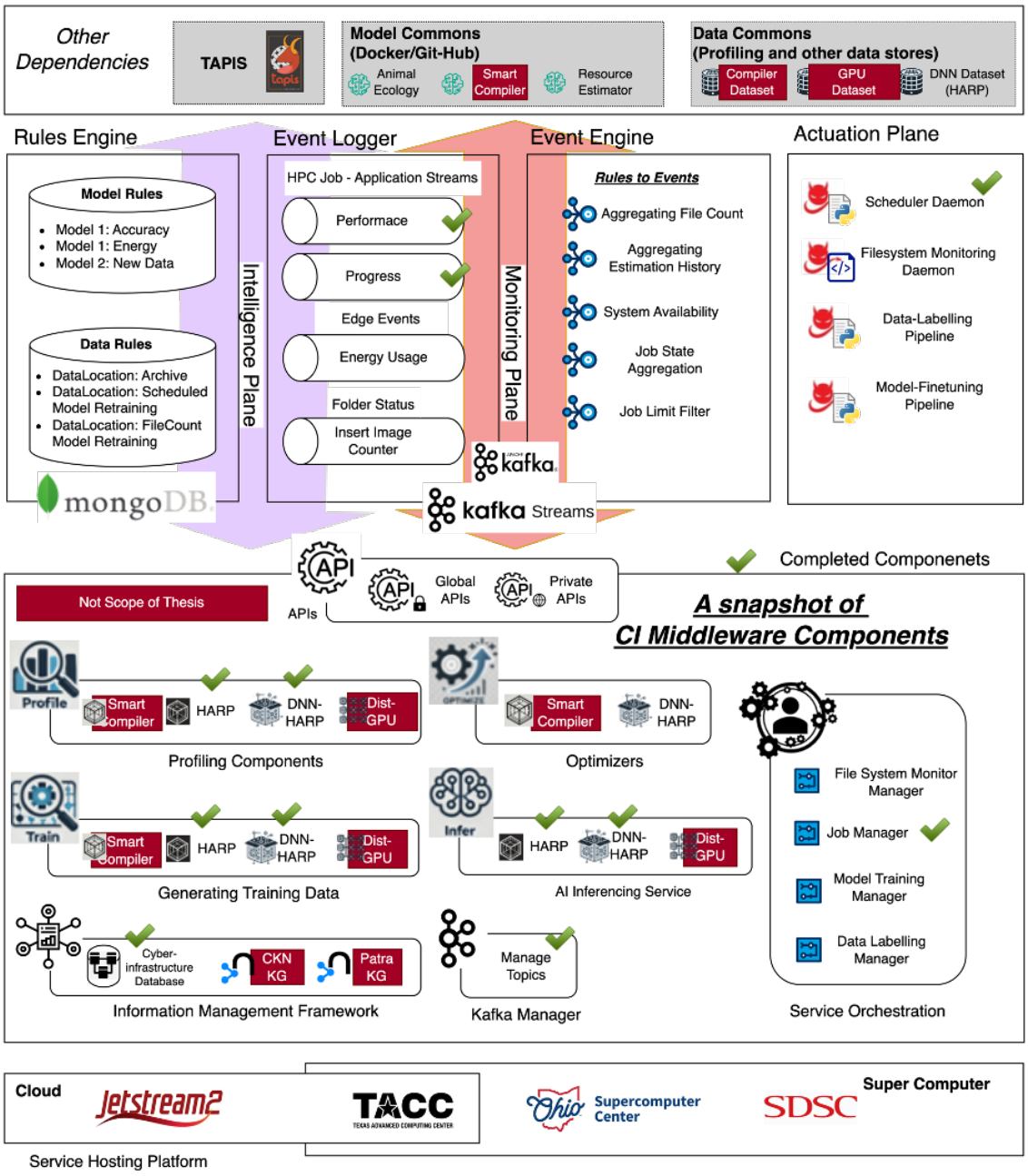


Figure 8.2: Overview of CI Middleware Components and Workflow in the ICICLE Intelligence Plane. The diagram illustrates key components, including data and model commons, monitoring and intelligence planes, profiling and optimization frameworks, and service orchestration across cloud and supercomputer environments.

- **Reasoning & Decision-Making** – Utilizes symbolic reasoning, reinforcement learning, or planning algorithms.
- **Action Execution** – Takes autonomous actions based on learned strategies. (Enabled via the **Service Orchestration Engine**)
- **Feedback Loop & Adaptation** – Continuously learns from past actions to improve future decisions. (Orchestrated via **CKN** and other monitoring tools in collaboration with the Rules Engine and trigger handlers such as the Edge Trigger Mechanism.)
- **Goal-Oriented Behavior** – Aligns actions with long-term objectives through self-improving mechanisms. (Defined by the Rules Engine, e.g., setting accuracy thresholds for the **Animal Ecology Use Case** or dynamically optimizing resource estimators.)

8.2.1 Use Cases

As part of future work, we aim to leverage the new Intelligence Plane - Agentic AI for MLOps to orchestrate an end-to-end workflow for animal ecology model training and deployment. This will involve integrating various ICICLE components while incorporating modular elements of HARP and iScheduler. By doing so, we seek to enable adaptive estimators that can generalize to new models or enhance accuracy based on historical data and available computational resources.

Use Case 1 - Animal Ecology

The invoker initiates the workflow by setting up the Edge Simulation, using the provided code repository from GitHub (https://github.com/tapis-project/camera-traps). The process begins with preparing the Edge Device Image and determining whether an image is out-of-distribution (OOD). If an OOD event is detected, it is stored in CKN and streamed via Kafka, where it undergoes rule-based filtering to assess if it meets OOD criteria.

Once the Monitoring Plane (MP) validates an OOD detection, it triggers the Retraining Event via CKN. At this stage, the Rule Engine reads predefined parameters to define necessary inputs for action. This includes:

- Selecting the appropriate model for retraining.
- Retrieving stored model weights.
- Identifying the location of labeled training datasets.
- Determining training parameters – either default settings or Hyperparameter Auto-Tuning (HPA).

Next, the Intelligence Plane (IP) assigns the retraining task to the Smart Scheduler (SS) Pipeline, which handles execution by:

1. Fetching available systems from the database.
2. Performing resource estimation using inference models and CKN inputs.
3. Checking queue availability via TAPIS.

4. Selecting the fastest system with optimal runtime and queue availability before submitting the job.

Once submitted, the Job Daemon continuously monitors execution, tracking job status, verifying if it will complete within the allocated time, and ensuring successful completion.

Upon completion, the Model Retraining Pipeline updates the trained model version, stores it on Hugging Face, and updates the corresponding Patra Model Cards for documentation and deployment. Throughout the process, Sachith and Neelesh oversee critical aspects of the Intelligence Plane's execution and model storage, ensuring efficient scheduling and deployment.

Use Case 2: HARP - Adaptive Estimation Refinement

In Chapter 6, we explored how the availability of new data (as shown in Figure 6.2) facilitates retraining estimators to adapt to new environments (e.g., a new cluster). Building on this, we leverage the existing HARP and Intelligence Plane (IP) framework (depicted in Figure 4.2, 4.3) to establish a workflow similar to the Animal Ecology use case. This workflow is triggered by estimation errors (e.g., misestimations for MegaDetector) to retrain the MegaDetector estimators and deploy updated models for improved inference in future executions.

Bibliography

[1] Nick Brown, Gordon Gibb, Evgenij Belikov, and Rupert Nash. Predicting batch queue job wait times for informed scheduling of urgent hpc workloads. *arXiv preprint arXiv:2204.13543*, 2022.

[2] Henri Casanova, Yick Ching Wong, Loïc Pottier, and Rafael Ferreira Da Silva. On the feasibility of simulation-driven portfolio scheduling for cyberinfrastructure runtime systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 3–24. Springer, 2022.

[3] Václav Chlumskỳ and Dalibor Klusáček. Improving accuracy of walltime estimates in pbs professional using soft walltimes. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 192–210. Springer, 2022.

[4] Qiyang Ding, Pengfei Zheng, Shreyas Kudari, Shivaram Venkataraman, and Zhao Zhang. Mirage: Towards low-interruption services on batch gpu clusters with reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[5] Lukas Ehrig, Daniel Atzberger, Benjamin Hagedorn, Jan Klimke, and Jürgen Döllner. Customizable asymmetric loss functions for machine learning-based predictive maintenance. In *2020 8th International Conference on Condition Monitoring and Diagnosis (CMD)*, pages 250–253. IEEE, 2020.

[6] Maurice Fallon, Hordur Johannsson, Michael Kaess, and John J Leonard. The mit stata center dataset. *The International Journal of Robotics Research*, 32(14):1695–1699, 2013.

[7] Harvard Medical School Research Computing. Smart slurm: Dynamic and accurate resource allocation for o2 jobs, 2024. Accessed: 2024-03-01.

[8] Marta Jaros and Jiri Jaros. Optimization of execution parameters of moldable ultrasound workflows under incomplete performance data. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 152–171. Springer, 2022.

[9] Morris A Jette and Tim Wickberg. Architecture of the slurm workload manager. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 3–23. Springer, 2023.

[10] David Kaufman and John Smith. A learned performance model for tensor processing units. *IEEE Transactions on Machine Learning*, 5(4), 2021.

[11] Dalibor Klusáček and Václav Chlumskỳ. Real-life hpc workload trace featuring refined job runtime estimates. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–19. Springer, 2024.

[12] Daiki Nakai, Keichi Takahashi, Yoichi Shimomura, and Hiroyuki Takizawa. A node selection method for on-demand job execution with considering deadline constraints. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 141–160. Springer, 2024.

[13] Tatsuyoshi Ohmura, Yoichi Shimomura, Ryusuke Egawa, and Hiroyuki Takizawa. Toward building a digital twin of job scheduling and power management on an hpc system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 47–67. Springer, 2022.

[14] Nwamaka Okafor, Bethany Lusch, and Venkatram Vishwanath. Queue wait time prediction in high performancecomputing (hpc) systems. 2024.

[15] Dhabaleswar K Panda, Vipin Chaudhary, Eric Fosler-Lussier, Raghu Machiraju, Amit Majumdar, Beth Plale, Rajiv Ramnath, Ponnuswamy Sadayappan, Neelima Savardekar, and Karen Tomko. Creating intelligent cyberinfrastructure for democratizing ai. *AI Magazine*, 45(1):22–28, 2024.

[16] Mangpo Phothilimthana, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, Michael Burrows, Charith Mendis, and Bryan Perozzi. Tpugraphs: A performance prediction dataset on large tensor computational graphs. *Advances in Neural Information Processing Systems*, 36:70355–70375, 2023.

[17] Anat Reiner-Benaim, Anna Grabarnick, and Edi Shmueli. Highly accurate prediction of jobs runtime classes. *arXiv preprint arXiv:1605.00388*, 2016.

[18] Joe Stubbs, Richard Cardone, Mike Packard, Anagha Jamthe, Smruti Padhy, Steve Terry, Julia Looney, Joseph Meiring, Steve Black, Maytal Dahan, et al. Tapis: An api platform for reproducible, distributed computational research. In *Advances in Information and Communication: Proceedings of the 2021 Future of Information and Communication Conference (FICC), Volume 1*, pages 878–900. Springer, 2021.

[19] Mohammed Tanash, Huichen Yang, Daniel Andresen, and William Hsu. Ensemble prediction of job resources to improve system performance for slurm-based hpc systems. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, pages 1–8. 2021.

[20] Vanamala Venkataswamy, Jake Grigsby, Andrew Grimshaw, and Yanjun Qi. Launchpad: Learning to schedule using offline and online rl methods. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 60–83. Springer, 2024.

[21] Sachith Withana and Beth Plale. Ckn: An edge ai distributed framework. In *2023 IEEE 19th International Conference on e-Science (e-Science)*, pages 1–10. IEEE, 2023.

[22] Wei Zhang and Yong Liu. Gpu performance modeling for deep learning workloads. *International Journal of Parallel Computing*, 47(3), 2023.