

**UNIVERSITY OF CYPRUS**  
**DEPARTMENT OF COMPUTER SCIENCE**

**INDIVIDUAL THESIS**

May 2026

Individual Diploma Thesis

**From Experiments to Insights: Automated Edge Benchmarking  
and Fingerprint-Driven Performance Modeling with BenchPilot**

Christos Symeou

**UNIVERSITY OF CYPRUS**



**DEPARTMENT OF COMPUTER SCIENCE**

May 2026

**UNIVERSITY OF CYPRUS**  
**DEPARTMENT OF COMPUTER SCIENCE**

**From Experiments to Insights: Automated Edge Benchmarking  
and Fingerprint-Driven Performance Modeling with BenchPilot**

**Christos Symeou**

Supervisor Professor  
George Pallis

The Individual Thesis was submitted for partial fulfilment of the requirements for obtaining a degree in Computer Science from the Department of Computer Science at the University of Cyprus.

May 2026

*I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (<https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/>), and with full accountability on my part for the accuracy, integrity, and validity of all content.”*

# Acknowledgments

I would like to express my sincere appreciation to my supervisor, Dr. George Pallis, for his guidance and support throughout the course of this thesis. His advice in shaping the direction of this work, as well as his feedback, contributed to the development of this thesis.

Moreover, I would like to extend my special thanks to Ph.D. candidate Joanna Georgiou for her continuous support throughout the thesis process. Her assistance in helping me understand and become familiar with the work conducted in the laboratory, along with our weekly meetings and her constructive feedback, played a significant role in the completion of this thesis.

Finally, I would like to thank my fellow students, who inspired and motivated me throughout this journey, as well as my friends and family for their continuous encouragement and psychological support.

# Abstract

Edge computing environments are increasingly used to support applications that require low latency and efficient use of limited resources by relocating computation closer to data sources [3]. However, these environments often operate under multi-tenant conditions, where several workloads share the same edge device and compete for common resources such as CPU, memory, storage, and network bandwidth [2]. This can lead to performance interference, making it difficult to predict how an application will behave when running alongside other workloads [13], [14]. Understanding these effects requires systematic experimentation, structured data collection, and tools that reduce the manual effort involved in managing experiments [1].

This thesis addresses these challenges through two complementary contributions. First, it develops a workload fingerprint extraction pipeline that transforms raw BenchPilot experiment outputs into structured records. Each fingerprint captures workload identity, hardware characteristics, application-level performance, and summarized resource utilization metrics, including CPU, memory, input/output, network, and power consumption. The generated fingerprints are organized into application-specific datasets for database performance, machine learning inference, streaming latency, and streaming tuples. These datasets are then used to train and evaluate several regression models for performance prediction, including Linear Regression, Decision Tree, Random Forest, Extra Trees, Gradient Boosting, AdaBoost, and XGBoost [17]–[24].

The evaluation shows that workload fingerprints provide a useful representation for modelling performance in multi-tenant edge environments. Tree-based and ensemble-based models generally outperform Linear Regression, indicating that workload performance is influenced by non-linear relationships between workload composition, hardware characteristics, and resource usage [26], [27]. Extra Trees achieves the best results for database performance and machine learning inference, Decision Tree performs best for streaming latency, and XGBoost performs best for streaming tuples. Feature importance analysis, using permutation importance [30] and SHAP interpretation [16],

further shows that different workload types depend on different resource dimensions. Database performance is mainly influenced by input/output and CPU features, machine learning inference by workload identity, power, and CPU usage, and streaming workloads mainly by network behaviour. Clustering analysis using KMeans [25] also reveals distinct resource usage profiles, including CPU-intensive, input/output-intensive, network-heavy, and high-power executions.

The second contribution of this thesis is the design and implementation of a web-based BenchPilot interface. The interface extends the original BenchPilot workflow [15] by integrating experiment configuration, YAML generation, deployment, monitoring, Docker service inspection, result access, validation, and Jupyter notebook integration into a single environment. Compared with the original workflow, the interface reduces manual file editing, terminal usage, output navigation, and notebook setup, while supporting more organized and reproducible experimentation.

Overall, this thesis demonstrates that workload fingerprints can support performance prediction and workload characterization in edge environments, while the developed BenchPilot interface improves the practicality of conducting large-scale and repeatable experiments. Together, these contributions provide a foundation for more efficient edge benchmarking and future performance-aware workload management.

# Contents

|                                                                                                    |           |
|----------------------------------------------------------------------------------------------------|-----------|
| <b>Chapter 1: Introduction .....</b>                                                               | <b>1</b>  |
| <b>1.1 Motivation .....</b>                                                                        | <b>1</b>  |
| <b>1.2 Contributions .....</b>                                                                     | <b>2</b>  |
| <b>1.2.1 Workload Fingerprint Extraction and Dataset Construction .....</b>                        | <b>2</b>  |
| <b>1.2.2 Machine Learning-Based Performance Modeling.....</b>                                      | <b>3</b>  |
| <b>1.2.3 Feature Importance and Resource Impact Analysis .....</b>                                 | <b>3</b>  |
| <b>1.2.4 Workload Characterization through Clustering .....</b>                                    | <b>3</b>  |
| <b>1.2.5 Analysis of Experimentation Limitations in BenchPilot .....</b>                           | <b>3</b>  |
| <b>1.2.6 Design and Implementation of a Web-Based BenchPilot Interface .....</b>                   | <b>4</b>  |
| <b>1.2.7 Integrated Experiment Management for Reproducible Edge<br/>        Benchmarking .....</b> | <b>4</b>  |
| <b>1.3 Structure of the Thesis .....</b>                                                           | <b>4</b>  |
| <b>Chapter 2: Background and Related Work .....</b>                                                | <b>6</b>  |
| <b>2.1 Edge Computing and Multi-Tenancy .....</b>                                                  | <b>6</b>  |
| <b>2.2 Benchmarking in Edge Computing .....</b>                                                    | <b>7</b>  |
| <b>2.3 BenchPilot Framework .....</b>                                                              | <b>9</b>  |
| <b>2.3.1 System Architecture and Experiment Declaration.....</b>                                   | <b>9</b>  |
| <b>2.3.2 Deployment Pipeline .....</b>                                                             | <b>11</b> |
| <b>2.3.3 Monitoring and Post-Experiment Analysis .....</b>                                         | <b>12</b> |
| <b>2.3.4 Limitations Relevant to This Thesis.....</b>                                              | <b>13</b> |
| <b>2.4 Workloads .....</b>                                                                         | <b>14</b> |
| <b>2.5 Workload Fingerprints and ML-based Performance Modeling .....</b>                           | <b>15</b> |
| <b>2.5.1 The fingerprint extraction pipeline .....</b>                                             | <b>16</b> |
| <b>2.5.2 Feature structure .....</b>                                                               | <b>17</b> |

|                                                          |    |
|----------------------------------------------------------|----|
| 2.5.3 ML-based performance prediction .....              | 18 |
| 2.6 Experimentation Challenges .....                     | 19 |
| 2.7 Summary and Research Gap .....                       | 20 |
| Chapter 3: Implementation.....                           | 22 |
| 3.1 Workload Fingerprint Extraction Pipeline .....       | 22 |
| 3.1.1 Fingerprint Extraction Module .....                | 23 |
| 3.1.2 Fingerprint Creation Module .....                  | 24 |
| 3.1.3 Dataset Construction .....                         | 26 |
| 3.2 Data Preparation and Performance Modeling .....      | 28 |
| 3.2.1 Data Preparation.....                              | 29 |
| 3.2.2 Regression Model Selection .....                   | 30 |
| 3.2.3 Model Evaluation Procedure .....                   | 31 |
| 3.3 Feature Importance .....                             | 32 |
| 3.3.1 Individual Feature Importance .....                | 33 |
| 3.3.2 Grouped Feature Importance .....                   | 34 |
| 3.4 Clustering .....                                     | 35 |
| 3.5 BenchPilot Interface Design and Implementation ..... | 37 |
| 3.5.1 Design Goals .....                                 | 37 |
| 3.5.2 System Architecture .....                          | 46 |
| 3.5.3 Backend Implementation .....                       | 48 |
| 3.5.4 Frontend Implementation .....                      | 54 |
| Chapter 4: Evaluation .....                              | 75 |
| 4.1 Dataset Overview .....                               | 75 |
| 4.2 Regression Model Performance .....                   | 78 |
| 4.2.1 Database Performance .....                         | 78 |
| 4.2.2 Machine Learning Inference Performance .....       | 79 |
| 4.2.3 Streaming Latency Performance.....                 | 80 |

|                                                                   |     |
|-------------------------------------------------------------------|-----|
| 4.2.4 Streaming Tuples Performance .....                          | 81  |
| 4.2.5 Summary of Regression Results .....                         | 82  |
| 4.3 Feature Importance Results .....                              | 83  |
| 4.3.1 Individual Feature Importance .....                         | 84  |
| 4.3.2 Grouped Feature Importance .....                            | 103 |
| 4.4 Clustering .....                                              | 116 |
| 4.4.1 Cluster Selection Using Elbow and Silhouette Analysis ..... | 117 |
| 4.4.2 KMeans Clustering and Centroid Interpretation .....         | 120 |
| 4.5 BenchPilot Interface Evaluation .....                         | 128 |
| 4.5.1 Original BenchPilot Workflow .....                          | 128 |
| 4.5.2 BenchPilot Interface Workflow .....                         | 133 |
| 4.5.3 Workflow Comparison .....                                   | 135 |
| 4.5.4 Summary .....                                               | 136 |
| Chapter 5: Conclusion .....                                       | 138 |
| 5.1 Summary of the Thesis .....                                   | 138 |
| 5.2 Main Findings .....                                           | 139 |
| 5.3 Limitations .....                                             | 140 |
| 5.4 Future Work .....                                             | 142 |
| 5.5 Final Remarks .....                                           | 144 |
| References .....                                                  | 145 |

# Chapter 1: Introduction

## 1.1 Motivation

As edge infrastructures continue to expand in both scale and heterogeneity, ensuring predictable performance is becoming an increasingly complex challenge [3], [5]. This difficulty is amplified in multi-tenant environments, where multiple workloads are co-located on the same device and compete for limited computational and energy resources [2], [8]. In such settings, even relatively minor resource contention can lead to noticeable degradations in latency, throughput, and energy efficiency, ultimately affecting the reliability of deployed applications [13], [14].

Despite the growing importance of this issue, many benchmarking frameworks remain reactive, capturing performance interference only after it has already occurred. Recent efforts have explored more data-driven evaluation in multi-tenant edge environments, but their effectiveness depends on the availability of large, diverse, and systematically collected datasets. In practice, generating such datasets is difficult because experimentation workflows often remain fragmented and manual, relying on configuration files, command-line deployment, and repetitive execution processes [1], [2], [15].

Motivated by the need for more proactive and data-driven performance management, this thesis investigates whether compact workload fingerprints can be used to model and predict interference patterns in edge environments. Although such representations provide a useful abstraction of workload behaviour, accurately capturing the interactions between co-located workloads and shared resources remains challenging. Addressing this challenge requires systematic data collection across different workloads, configurations, and devices [2], [13], [14].

To this end, experiments were conducted using the BenchPilot framework, which supports automated and repeatable benchmarking in edge environments [1], [15].

However, the experimentation process still introduces significant overhead, as tasks such as defining experiments, managing configuration files, deploying workloads, monitoring execution, and debugging failed runs require manual effort. These limitations become more visible when experiments must be repeated across many workload combinations and devices.

To address this gap, this thesis extends BenchPilot by introducing a unified interface for experiment configuration, deployment, monitoring, and result management. By streamlining the experimentation workflow, the proposed solution reduces manual effort, improves reproducibility, and enables more scalable data collection. This supports more effective performance analysis and provides a foundation for future research on co-location-aware and predictive scheduling in edge environments [2].

## **1.2 Contributions**

Based on the challenges and objectives described above, this thesis contributes to both the analysis of workload performance and the improvement of the experimentation process. On the analysis side, it uses workload fingerprints and machine learning models to study performance behaviour in multi-tenant edge environments. On the system side, it extends BenchPilot with a unified interface that simplifies experiment configuration, deployment, monitoring, and result management.

The main contributions of the thesis are summarized below.

### **1.2.1 Workload Fingerprint Extraction and Dataset Construction**

An automated pipeline was designed and implemented to extract workload fingerprints from raw experimental data collected through the BenchPilot framework [1]. Each fingerprint captures workload identity, hardware characteristics, observed performance, and summarized resource utilization metrics, including CPU, memory, network, input/output, and power consumption. The generated fingerprints are then organized into application-specific datasets for database performance, machine learning inference,

streaming latency, and streaming tuples, providing the structured data used for performance modelling and workload analysis.

### **1.2.2 Machine Learning-Based Performance Modeling**

Using the constructed fingerprint datasets, several regression models were trained and evaluated to predict workload performance under different execution and co-location conditions. The models were used to examine whether fingerprint features can capture the relationship between workload composition, hardware characteristics, resource utilization, and application-level performance. This contribution supports the study of performance behaviour in multi-tenant edge environments.

### **1.2.3 Feature Importance and Resource Impact Analysis**

To better understand the factors influencing performance, feature importance analysis was applied to the trained models. Both individual feature importance and grouped resource analysis were used to identify which workload characteristics and resource categories had the strongest effect on performance prediction. SHAP-based interpretation was also used to provide a more detailed view of model behaviour [16].

### **1.2.4 Workload Characterization through Clustering**

Unsupervised learning was applied to group workload executions according to their resource utilization patterns. This analysis helped identify workload profiles such as CPU-intensive, input/output-intensive, network-heavy, and high-power executions. The clustering results provide an additional way to understand workload behaviour beyond regression-based prediction.

### **1.2.5 Analysis of Experimentation Limitations in BenchPilot**

A detailed analysis of the original BenchPilot workflow was conducted to identify practical limitations related to experiment configuration, deployment, monitoring, result retrieval, and post-experiment analysis [1], [15]. This analysis showed that several stages

of the workflow remain manual and split across multiple tools, motivating the need for a more integrated and user-friendly experimentation system.

### **1.2.6 Design and Implementation of a Web-Based BenchPilot Interface**

To address the identified workflow limitations, a web-based interface was designed and implemented to extend BenchPilot. The interface supports user authentication, cluster and workload management, YAML generation, individual and batch experiment deployment, execution monitoring, Docker service inspection, result access, validation messages, comments, review states, and Jupyter notebook integration. This reduces reliance on manual file editing, terminal commands, and direct navigation through output directories.

### **1.2.7 Integrated Experiment Management for Reproducible Edge Benchmarking**

The proposed interface supports the full lifecycle of experimentation, including configuration, deployment, monitoring, and result retrieval. By integrating these stages into a cohesive workflow, the system simplifies experiment management, reduces the potential for configuration errors, and improves usability for both experienced and novice users.

## **1.3 Structure of the Thesis**

The remainder of this thesis is organized as follows. Chapter 2 provides the background and related work needed to understand the problem addressed in this thesis. It introduces edge computing and multi-tenancy, discusses benchmarking in edge environments, and presents the BenchPilot framework. The chapter also describes the workloads used in the experiments, introduces workload fingerprints and machine learning-based performance modelling, and explains the experimentation challenges that motivate the proposed work.

Chapter 3 presents the implementation of the proposed approach. It first describes the workload fingerprint extraction pipeline, including fingerprint extraction, fingerprint creation, and dataset construction. It then explains the data preparation process, the regression models used for performance prediction, and the feature importance and

clustering methods applied for workload analysis. The chapter also presents the design and implementation of the BenchPilot interface, including its design goals, system architecture, backend implementation, and frontend functionality.

Chapter 4 evaluates both parts of the thesis. It first examines the generated fingerprint datasets and evaluates the performance of the regression models for database performance, machine learning inference, streaming latency, and streaming tuples. It then analyzes the individual and grouped feature importance results and discusses the clustering results used to characterize workload behaviour. Finally, the chapter evaluates the BenchPilot interface by comparing the original BenchPilot workflow with the developed interface workflow.

Chapter 5 concludes the thesis by summarizing the work carried out, presenting the main findings, discussing the limitations of the study, and outlining possible directions for future work.

## Chapter 2: Background and Related Work

### 2.1 Edge Computing and Multi-Tenancy

Edge computing extends traditional cloud computing by moving computation closer to where data is generated. Instead of transmitting all data from Internet of Things (IoT) devices to centralized cloud data centers, part of the processing is performed on intermediate nodes such as gateways, base stations, or edge servers. This paradigm shift is largely driven by the rapid growth of IoT, with projections estimating up to 500 billion connected devices by 2030 [2]. Transmitting large volumes of data to distant cloud infrastructures introduces communication delays, increases bandwidth consumption, and can negatively impact the reliability of latency-sensitive applications [3]. By relocating computation closer to data sources, edge computing reduces these overheads and enables faster, more responsive services [3].

Within such environments, multi-tenancy emerges as a fundamental design principle. Multi-tenancy refers to the ability of a system to support multiple users or applications on shared physical infrastructure while maintaining logical isolation. At the edge, this model is often necessary rather than optional, as deploying dedicated hardware for each application is impractical due to cost, energy, and resource constraints [7], [8]. Consequently, multiple workloads are typically co-located on the same device, sharing its computational and communication resources.

This is primarily enabled through lightweight virtualization technologies, particularly containerization. Containers encapsulate applications and their dependencies within isolated environments while sharing the host operating system kernel. Compared to virtual machines, containers introduce significantly lower overhead and faster startup times, making them well suited for heterogeneous and resource-constrained edge devices [7], [9]. Platforms such as Docker provide standardized mechanisms for packaging,

deploying, and managing containerized applications, facilitating efficient resource utilization and scalable multi-tenant deployments [9], [14].

Despite its advantages, multi-tenancy introduces important performance challenges. When multiple workloads share the same device, they compete for common resources including CPU, memory, storage, and network bandwidth. This competition can lead to performance interference, commonly referred to as co-location interference, where the behaviour of one workload affects the performance of others. In practice, this may appear as increased latency, reduced throughput, and lower predictability compared to isolated execution. While such effects have been extensively studied in cloud and containerized environments [13], [14], they are often more pronounced in edge settings due to limited resource availability and hardware heterogeneity [2]. Recent studies on multi-tenant edge systems further highlight that understanding these interactions requires systematic experimentation across diverse workload combinations [2].

These challenges underline the need for benchmarking frameworks capable of evaluating multi-tenant performance in realistic edge environments. Tools such as BenchPilot support this by automating the deployment of containerized workloads and the collection of performance metrics on heterogeneous edge infrastructures [1]. However, as discussed in later sections, existing workflows still require considerable manual effort and are often split across multiple tools, limiting scalability. This motivates the need for more automated and integrated approaches, which form a key focus of this thesis.

## **2.2 Benchmarking in Edge Computing**

Benchmarking is the systematic process of executing controlled experiments on a system under test and observing its behavior through performance metrics [1]. Instead of relying only on hardware specifications, benchmarking uses representative workloads to measure quantities such as latency, throughput, resource utilization, and energy consumption. This allows researchers and engineers to evaluate how a system behaves in practice under well-defined conditions.

Benchmarking is important for several reasons. First, performance cannot always be predicted from specifications alone, since systems with similar hardware may behave differently when executing the same workload. Second, benchmarking provides objective data for comparing alternative configurations, technologies, or deployment choices. Third, by revealing bottlenecks and performance trade-offs, benchmarking supports optimization and improves reproducibility, since experiments can be repeated under the same conditions.

In edge computing, benchmarking becomes both more important and more challenging. Edge infrastructures are heterogeneous, often limited in available resources, and commonly host multiple workloads on the same device. This can lead to co-location interference, as discussed in the previous section. As a result, evaluating a single workload on a single device is not sufficient. Therefore, benchmarking in edge environments needs to capture how different workload combinations behave across different devices and configurations.

To support this need, frameworks such as BenchPilot were introduced to automate the deployment of containerized workloads, collect performance and infrastructure metrics, and enable repeatable experiments on real edge micro-datacenters [1], [15]. However, when many multi-tenant scenarios must be explored, the experimentation workflow can still involve considerable manual effort. This limitation motivates the extensions and automation developed in this thesis.

## 2.3 BenchPilot Framework

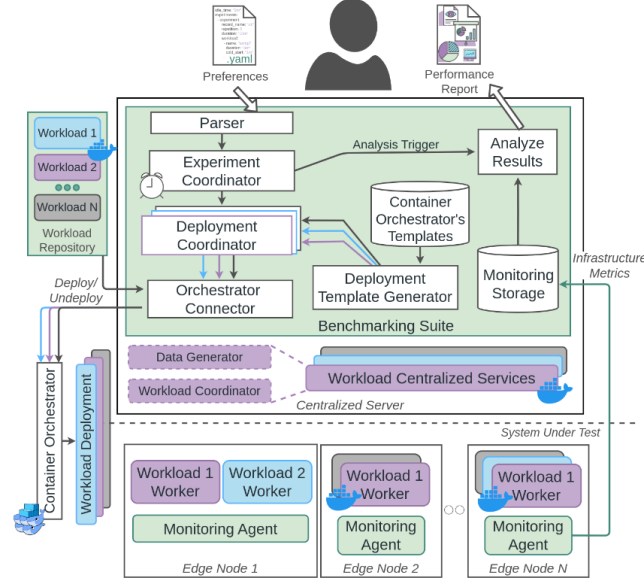


Figure 1. Overview of the BenchPilot framework architecture.

BenchPilot [1] is an open-source benchmarking framework designed for edge micro-datacenters. It supports repeatable experiments by helping users define edge clusters, deploy containerized workloads, collect performance metrics, and analyze results. This makes it useful for studying heterogeneous edge environments, where devices may differ in hardware capabilities and workloads may have different resource demands [1], [15].

### 2.3.1 System Architecture and Experiment Declaration

At the heart of BenchPilot is a high-level declarative model that allows users to describe their experiments using structured YAML-based configuration files. Before experimentation begins, the user defines two things, the cluster under test and the experiments to be performed. The cluster is specified in a dedicated configuration file, where each node is described by its IP address, hostname, and authentication credentials, either a username and password or an SSH key path. Once the cluster is defined,

BenchPilot handles the bootstrapping process automatically, installing Docker, Docker Compose, and all required dependencies on each worker node without requiring any manual intervention from the user [15].

Experiments are then described in a separate YAML file following BenchPilot's experiment model. Each experiment entry includes a record name used for later metric retrieval, the number of repetitions to ensure reproducibility, the total experiment duration, and one or more workload definitions to be deployed concurrently on the target nodes. Each workload specifies its name, the cluster nodes it will run on, the container orchestrator to be used with Docker Swarm as the default, and any workload-specific parameters such as data injection rates, thread counts, or engine configurations. An idle period between consecutive experiments can also be configured, allowing the system to return to a stable baseline state before the next run begins [1], [15].

Figure 2 shows a representative configuration used during the experiments conducted in this thesis, where three workloads, namely a machine learning inference task, a streaming analytics pipeline, and a NoSQL database workload, are co-located simultaneously on the same Raspberry Pi node. This example illustrates how BenchPilot's declarative model enables complex multi-workload co-location scenarios to be expressed concisely, without requiring any modifications to the underlying workload code or manual orchestration steps.

```

1 idle_between_experiments: "2m"
2 experiments:
3   - experiment:
4     record_name: "mlN-stm-mdb_rpi5"
5     repetition: 1
6     duration: "default"
7     workloads:
8       - name: "mlperf"
9         cluster: ["raspberrypi5"]
10        orchestrator: "swarm"
11        duration: "20m"
12        parameters:
13          dataset_folder: "imagenet2012"
14          model_file: "resnet50_v1"
15          profile: "resnet50-ncnn"
16          data_volume_path: "/mlperf/data"
17          model_volume_path: "/mlperf/model"
18          output_volume_path: "/mlperf/output"
19          worker_threads: 4
20
21       - name: "marketing-campaign"
22         cluster: ["raspberrypi5"] # workers' hostnames
23         duration: "20m"
24         parameters:
25           num_of_campaigns: 100000000
26           capacity_per_window: 10000000
27           kafka_event_count: 100000000000
28           time_divisor: 10000000
29           workload_tuples_per_second_emission: 1000000000
30           engine:
31             name: "storm"
32             parameters:
33               ackers: 2
34               executors_per_node: [ 4 ]
35               ui_port: "8080"
36
37       - name: "database"
38         duration: "20m"
39         cluster: ["raspberrypi5"]
40         parameters:
41           db: "mongodb"
42           threads: 4

```

Figure 1. YAML experiment configuration showing co-located mlperf, marketing-campaign, and database workloads on raspberrypi5

### 2.3.2 Deployment Pipeline

Once an experiment description is submitted, BenchPilot processes it through a structured internal pipeline. A parser first validates the configuration for syntactic correctness and, if no errors are detected, passes each experiment sequentially to the Deployment Template Generator. This component transforms the user's declared preferences into Docker Compose templates that are compatible with the underlying container orchestrator. The Deployment Coordinator then takes these templates and manages the complete execution lifecycle, deploying services to the target nodes, monitoring their operational health, recording the start and end timestamps of each experiment, and tearing down the deployment once execution is complete. The recorded timestamps are later used during the post-experiment analysis phase to retrieve the monitoring data corresponding to each individual run [1], [15].

### 2.3.3 Monitoring and Post-Experiment Analysis

BenchPilot includes an integrated monitoring and analysis workflow that supports the collection, storage, and examination of performance data throughout the experimentation process [1], [15]. Its monitoring system is designed to be transparent and non-intrusive, allowing workloads to execute without interference. During the initial setup phase, a Netdata agent is deployed as a container on each worker node to collect system-level metrics from performance-related files and cgroup data. The collected metrics are periodically gathered by a centralized Prometheus server and stored in InfluxDB for time-series storage and later analysis [1], [15].

Beyond system-level monitoring, BenchPilot supports detailed infrastructure and workload measurements. Energy consumption can be monitored through external smart plugs, while cAdvisor captures container-level resource usage. In this way, the framework records key metrics such as CPU utilization, memory usage, disk activity, network traffic, power consumption, and CPU temperature, alongside application-level metrics such as throughput and latency [1], [15]. These measurements provide a combined view of how workloads and resources behave under different experimental conditions.

Once experiments are completed, BenchPilot enables post-experiment analysis through a Jupyter Notebook environment [15]. Using the `PostExecuteController` class from the BenchPilotSDK, users query Prometheus with the recorded start and end timestamps of each experiment to retrieve the corresponding monitoring data. The retrieved metrics can be exported to CSV files or analyzed directly in the notebook using custom scripts, models, and visualizations [15]. This workflow allows researchers to move from raw monitoring data to performance insights within the same ecosystem, but still requires users to manage configuration, execution, data retrieval, and analysis through separate steps and interfaces [1], [15]. However, as discussed in later sections, existing workflows still require considerable manual effort and are often split across several tools and steps, limiting scalability. This motivates the need for more automated and integrated approaches, which form a key focus of this thesis.

### 2.3.4 Limitations Relevant to This Thesis

Despite its capabilities, the current BenchPilot workflow presents several practical limitations that became evident during the data collection phase of this thesis. While the framework provides a Jupyter-based interface that allows users to navigate files, create and edit experiment configurations, and access results, the execution of experiments still relies on command-line interaction [1], [15]. Initiating deployments, restarting failed runs, and managing experiment lifecycles require users to switch from the notebook environment to the terminal, increasing complexity and fragmenting the workflow.

This separation between configuration and execution also affects monitoring and post-experiment analysis. Although BenchPilot automatically collects rich infrastructure and application metrics, retrieving and analyzing them requires manual coordination of several steps. Users must ensure that experiments have completed, locate the correct timestamps, invoke the PostExperimentController, and export metrics to CSV files before applying custom analysis [1], [15]. As the number of experiments grows, repeatedly performing these actions becomes tedious and increases the risk of mistakes, such as misaligned time windows or misplaced result files.

Several other aspects of the experimentation workflow remain manual and error prone. Users must define directories for YAML configuration files, specify output locations for generated CSVs, and repeatedly create or modify configurations to explore different scenarios. When many workloads, parameter settings, and co-location combinations are involved, this process becomes time-consuming and difficult to manage consistently. In practice, these limitations restrict the scalability and reproducibility of large experimental campaigns.

Overall, the lack of an integrated interface for configuring, deploying, monitoring, and analysing experiments in a unified manner limits the usability of BenchPilot for extensive multi-tenant studies. This thesis addresses these limitations by introducing a unified interface that automates key steps in experiment configuration, deployment, monitoring integration, and result management, thereby simplifying the experimentation workflow and improving reproducibility.

## 2.4 Workloads

BenchPilot supports a diverse set of containerized workloads designed to capture different application behaviors and stress various system resources in edge environments. These workloads represent common edge computing scenarios, including streaming data processing, machine learning inference, database operations, and synthetic resource stressors. By combining these workloads, the framework enables the evaluation of both isolated execution and co-location scenarios, which is essential for analysing performance interference.

The streaming workload is based on the Yahoo Streaming Benchmark and represents a real-time data processing pipeline. It continuously processes incoming data through multiple stages, including filtering, aggregation, and storage, making it well suited for evaluating applications that are sensitive to latency. Machine learning workloads are derived from MLPerf and focus on inference tasks such as image classification. These workloads can run either independently or in a streaming mode, where data is received over the network, allowing both computational performance and communication overhead to be examined.

The database workload is based on the Yahoo Cloud Serving Benchmark and is used to evaluate the performance of NoSQL systems under continuous read, update, and insert operations. It provides insight into how data-driven applications behave under sustained load. In addition, BenchPilot includes synthetic stress workloads implemented using tools such as stress and IPerf3. These workloads are designed to target specific system resources, including CPU, memory, disk operations, and network bandwidth. They are particularly useful for controlled experiments, where isolating and understanding the effects of resource contention is essential.

To standardize experiment configuration and analysis, each workload follows a naming convention and is associated with specific performance metrics and target resources. Table 1 summarizes the workloads used in this thesis.

| Short Name | Application                                    | Description                                                | Collected Metric          | Service Name  | Target Resource |
|------------|------------------------------------------------|------------------------------------------------------------|---------------------------|---------------|-----------------|
| stm        | Marketing Campaign (Yahoo Streaming Benchmark) | Streaming data processing pipeline for real-time analytics | Application latency       | engine-worker | CPU, Network    |
| mdb        | MongoDB Database                               | NoSQL database executing continuous CRUD operations        | Total operations executed | database      | CPU             |
| mlN        | MLPerf (NCNN)                                  | Machine learning inference workload (local execution)      | Total queries processed   | MLPerf_worker | CPU             |
| mlO        | MLPerf (ONNX)                                  | Machine learning inference workload (local execution)      | Total queries processed   | MLPerf_worker | CPU             |
| mlTF       | MLPerf (TensorFlow)                            | Machine learning inference workload (local execution)      | Total queries processed   | MLPerf_worker | CPU             |
| mlNStr     | MLPerf (NCNN, streaming mode)                  | Inference workload receiving input over the network        | Total queries processed   | MLPerf_worker | CPU, Network    |
| mlOStr     | MLPerf (ONNX, streaming mode)                  | Inference workload receiving input over the network        | Total queries processed   | MLPerf_worker | CPU, Network    |
| mlTFStr    | MLPerf (TensorFlow, streaming mode)            | Inference workload receiving input over the network        | Total queries processed   | MLPerf_worker | CPU, Network    |
| IP3        | IPerf3                                         | Network bandwidth stress workload                          | -                         | simple        | Network         |
| scpu       | CPU Stress                                     | Synthetic workload stressing CPU resources                 | -                         | simple        | CPU             |
| svm        | Memory Stress                                  | Synthetic workload stressing memory usage                  | -                         | simple        | CPU, Memory     |
| sio        | Disk I/O Stress                                | Synthetic workload stressing storage operations            | -                         | simple        | CPU, Disk       |

Table 1. BenchPilot Workloads, Metrics, and Targeted Resources

## 2.5 Workload Fingerprints and ML-based Performance Modeling

Workload fingerprints provide a structured way to represent the behaviour of an experiment. Instead of analysing raw monitoring data directly, each experiment is summarized into a compact record that combines workload information, hardware characteristics, performance results, and resource utilization metrics. In this thesis, these fingerprints are used to study how different workload combinations and system conditions affect performance in multi-tenant edge environments.

A workload fingerprint includes the identity of the active workloads, the specifications of the device on which the experiment was executed, the observed application-level performance, and summarized resource metrics such as input/output, memory usage, network activity, CPU usage, and power consumption. By combining these elements into a single structured representation, fingerprints make it possible to compare experiments more consistently and prepare the data for machine learning-based analysis.

These fingerprints form the foundation for the performance modelling process used in this thesis. Machine learning models are trained on the generated fingerprint datasets to examine whether performance can be predicted from workload composition, hardware characteristics, and resource behaviour. In addition, feature importance and clustering are used to better understand which resources influence performance and how workload executions can be grouped according to similar utilization patterns.

### 2.5.1 The fingerprint extraction pipeline

The fingerprint extraction pipeline, illustrated in Figure 3, processes the data collected from BenchPilot experiments and converts it into structured workload fingerprint records. The pipeline is organized into two main modules: the Fingerprint Extraction Module and the Fingerprint Creation Module.

The Fingerprint Extraction Module links application-level performance results with the corresponding utilization metrics. It reads performance metrics from CSV files associated with different workload categories, including database workloads, machine learning inference workloads, and streaming analytics workloads. For each recorded experiment, it identifies the utilization metrics file that describes the resource usage observed during execution.

After the correct performance and utilization data have been matched, the Fingerprint Creation Module constructs the fingerprint record. It extracts resource metrics such as CPU usage, memory consumption, disk activity, network traffic, and power usage. For each resource category, it computes summary statistics, including the peak value, mean value, and a fluctuation measure based on the coefficient of variation.

The module also adds workload composition and hardware information to the fingerprint. Workload composition is represented as a binary vector indicating which workloads were active during the experiment. Hardware information includes device characteristics such as the number of cores, processor frequency, memory capacity, and storage.

All generated fingerprints are stored incrementally in a JSON file, forming a dataset of 897 records collected across multiple devices and workload combinations. This dataset is then converted into CSV format, producing the structured feature matrix used for performance modelling, feature importance analysis, and clustering.

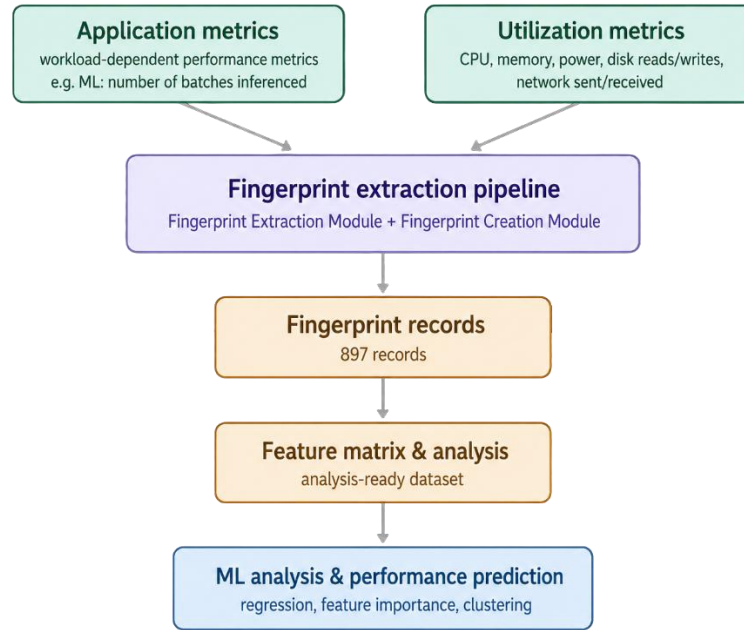


*Figure 3. Overview of the workload fingerprint extraction and analysis pipeline*

### 2.5.2 Feature structure

Each fingerprint record contains 36 features in total, organized into four main categories. These include workload identity features, represented as 12 binary indicators showing which workloads were active, hardware specification features describing the device characteristics, and resource utilization features that capture peak, mean, and fluctuation values across five resource dimensions. In addition, each fingerprint includes a performance value that serves as the target variable.

An example of a generated fingerprint is illustrated in Figure 4, which shows how these features are combined into a structured record. This structured format makes it possible to use the dataset in machine learning models, where the goal is to learn how different workloads and system conditions affect performance.

```

{
  "Application Metric": "db.csv",
  "Workload": "ML-O_DB_VM",
  "Performance": 580.156,
  "Filename": "ml0_mdb_svm_rpi0.csv",
  "Resource": "rpi0",
  "cores": 4,
  "frequency": 1.5,
  "ram": 4,
  "storage": 29,
  "scpu": 0,
  "mdb": 1,
  "sio": 0,
  "mlN": 0,
  "mlO": 1,
  "mlTF": 0,
  "mlNStr": 0,
  "mlOStr": 0,
  "mlTFStr": 0,
  "IP3": 0,
  "stm": 0,
  "svm": 1,
  "IO_peak": 36417.28,
  "IO_mean": 10885.8327,
  "IO_fluctuation": 0.9745176238147907,
  "Memory_peak": 3028.671,
  "Memory_mean": 1121.4102,
  "Memory_fluctuation": 0.27455146072338915,
  "Network_peak": 806.5898,
  "Network_mean": 166.1814,
  "Network_fluctuation": 0.9320435763495756,
  "CPU_peak": 100.0,
  "CPU_mean": 91.7217,
  "CPU_fluctuation": 0.11189610611178408,
  "Power_peak": 5.335,
  "Power_mean": 4.6162,
  "Power_fluctuation": 0.11732081648214585
},

```

Figure 4. Example of a workload fingerprint record

### 2.5.3 ML-based performance prediction

Using the extracted fingerprint dataset, machine learning techniques were applied to predict workload performance under different execution conditions. The goal is to understand how workload combinations, system characteristics, and resource utilization affect performance in edge environments.

Before training the models, the dataset was pre-processed by removing non-relevant features, encoding categorical values, and splitting the data into training and testing sets. Several regression models were then evaluated, including Linear Regression, Decision Trees, Random Forest, Extra Trees, Gradient Boosting, HistGradientBoosting, AdaBoost, and XGBoost [17]–[24].

Model performance was assessed using standard metrics, including  $R^2$  and Mean Absolute Error (MAE), across different application types such as database workloads, machine learning inference, and streaming applications.

To better understand the results, feature importance analysis was performed. Permutation importance was used to evaluate the impact of individual features, while grouped feature analysis provided a higher-level view by examining resource categories such as CPU, memory, network, input/output, power, and workload identity. In addition, SHAP analysis was used for selected models to provide more detailed explanations of feature contributions.

Clustering techniques were also applied to identify patterns in workload behavior. Using KMeans, workloads with similar resource usage characteristics were grouped together, allowing further analysis of their behavior and potential compatibility [25].

## 2.6 Experimentation Challenges

During the data collection phase of this thesis, several practical challenges became evident in the original BenchPilot workflow. Although BenchPilot supports repeatable benchmarking, users still need to coordinate experiment configuration, deployment, monitoring, and result retrieval across several tools and steps. This becomes increasingly difficult when many workloads, devices, and co-location scenarios must be tested.

For example, deploying an experiment such as `mdb-stm-scpu` requires the user to prepare the experiment configuration, define the correct output paths, create or update the YAML file, and launch the experiment from the terminal. This experiment represents the co-location of three workloads: `mdb`, which refers to the MongoDB database workload, `stm`, which refers to the streaming analytics workload based on the Yahoo Streaming Benchmark, and `scpu`, which refers to a CPU stress workload. Even for this single case, the user must ensure that the experiment name, workload configuration, node selection, and output locations are all consistent before deployment.

Monitoring also requires manual effort. The user must inspect execution output separately and use Docker commands to verify whether the deployed services are running correctly. If an experiment fails, remaining Docker services may need to be removed manually before another deployment can begin. In addition, the original workflow does not provide a unified view of the current system state or a built-in mechanism to determine whether another user is already running an experiment on the same device. This increases the risk of resource conflicts and unreliable results.

Managing the experimental infrastructure introduces further complexity. Adding a new device or modifying an existing node requires direct editing of the BenchPilot cluster configuration file. This process must be performed carefully to avoid configuration errors. Furthermore, experiments are launched individually in the original workflow, which makes large experimental campaigns time-consuming when many workload combinations must be executed across multiple devices.

These challenges motivated the development of the BenchPilot interface presented in this thesis. The interface addresses these limitations by providing authenticated user access, web-based experiment configuration, automatic YAML generation, individual and batch deployment, conflict detection, experiment monitoring, Docker service visibility, result access, and Jupyter notebook integration. In this way, the interface reduces manual effort and supports more organized and reproducible experimentation in multi-tenant edge environments.

## **2.7 Summary and Research Gap**

This chapter provided the background needed to understand the challenges addressed in this thesis. It discussed the characteristics of edge computing environments, the role of multi-tenancy, and the performance issues that arise when workloads are co-located. It also highlighted the importance of benchmarking for analysing these effects, introduced the BenchPilot framework as a tool for repeatable experimentation, and presented workload fingerprints as a structured way to capture workload behaviour for machine learning based analysis.

Despite this progress, there is still a clear gap between what is needed and what current approaches provide. While fingerprint-based modeling offers a promising way to understand and predict performance, it relies on collecting large amounts of consistent and well-organized data. In practice, however, the experimentation process in BenchPilot remains manual and fragmented, requiring users to manage configurations, execution, and results across different tools and interfaces. This makes experimentation time-consuming, harder to scale, and more prone to errors.

This thesis addresses this gap by extending BenchPilot with a unified interface that simplifies experiment configuration, deployment, and monitoring. By improving the experimentation workflow, the proposed approach supports more efficient data collection and enables more effective performance analysis. The next chapter presents the design and implementation of the system, including the workload fingerprint extraction pipeline, the machine learning models used for performance prediction, and the development of the BenchPilot interface.

## Chapter 3: Implementation

### 3.1 Workload Fingerprint Extraction Pipeline

As discussed in Section 2.5.1, the workload fingerprint extraction pipeline is implemented through two main modules: the Fingerprint Extraction Module and the Fingerprint Creation Module. The Fingerprint Extraction Module is responsible for reading the application-level performance metrics generated by each experiment. These metrics depend on the workload type. For example, machine learning workloads are evaluated based on the number of processed inference batches, while database workloads are evaluated based on the number of executed operations. The module then matches each performance record with the corresponding experiment and utilization metrics.

The Fingerprint Creation Module performs the core processing step. It extracts the relevant resource utilization metrics for each workload, including CPU usage, memory consumption, disk activity, network traffic, and power usage. For each resource category, it computes summary statistics such as peak value, mean value, and fluctuation. In addition, each fingerprint includes information about the active workload combination and the hardware characteristics of the device, such as the number of cores, processor frequency, memory capacity, and storage. All generated fingerprints are stored in a JSON file. The structure of a workload fingerprint is illustrated in Figure 5.

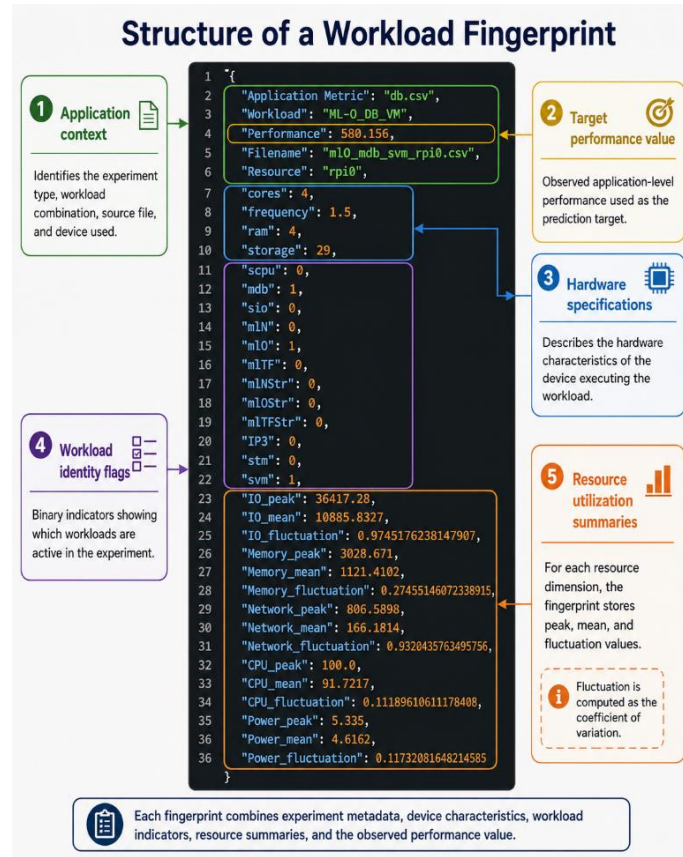


Figure 5. Structure of a workload fingerprint record and its main feature groups

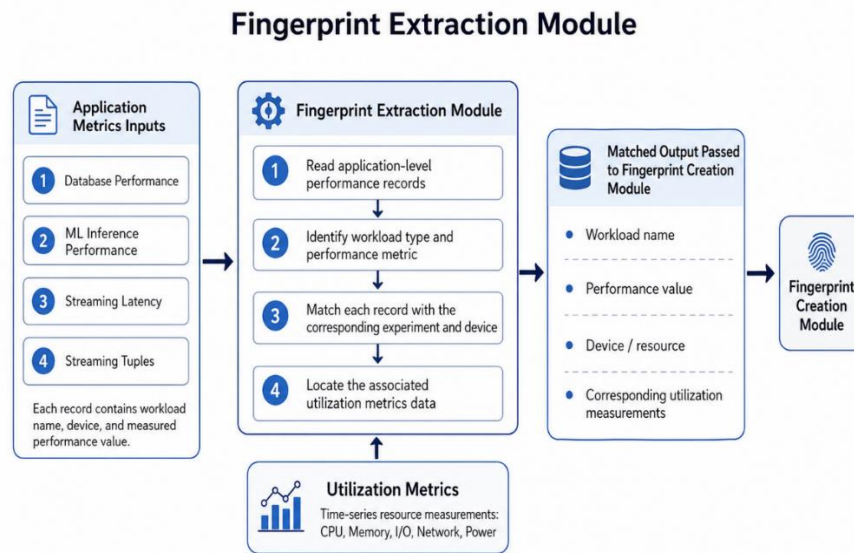
### 3.1.1 Fingerprint Extraction Module

The Fingerprint Extraction Module, illustrated in Figure 6, is responsible for linking the application-level performance results with the corresponding utilization metrics collected during each experiment. It processes the available application metric categories, namely database performance, machine learning inference performance, streaming latency, and streaming tuples. Each category reflects a different way of measuring workload behavior depending on the type of application under evaluation. For instance, database workloads are assessed based on the number of executed operations, machine learning workloads are evaluated based on the number of processed inference queries or batches, while streaming workloads are measured either in terms of latency or the number of processed tuples.

For each application metric category, the module reads the workload name together with the recorded performance value for a given device. It then identifies the utilization metrics

that correspond to the same experiment and device. These utilization metrics consist of time series measurements describing the resource usage observed during execution, including CPU usage, memory consumption, input/output activity, network traffic, and power consumption.

Once the appropriate performance and utilization data have been matched, the Fingerprint Extraction Module forwards the relevant information to the Fingerprint Creation Module. This includes the workload name, the measured performance value, the device on which the experiment was executed, and the associated utilization measurements. In this way, the module serves as the coordination layer of the pipeline, ensuring that each workload fingerprint is generated from the correct combination of application-level performance data and system-level utilization data.



The Fingerprint Extraction Module acts as the orchestration layer, linking application-level performance results with the correct system utilization data before fingerprint generation.

*Figure 6. Fingerprint Extraction Module*

### 3.1.2 Fingerprint Creation Module

The Fingerprint Creation Module, illustrated in Figure 7, is responsible for constructing the actual workload fingerprint. It receives the matched information from the Fingerprint

Extraction Module and converts it into a structured record that can later be used for analysis and machine learning-based modelling.

As shown in Figure 7, the module first adds the general experiment information to the fingerprint. This includes the application metric, the workload name, the observed performance value, the source filename, and the resource on which the workload was executed. It then enriches the fingerprint with the hardware characteristics of the device, including the number of CPU cores, processor frequency, RAM capacity, and storage capacity. These features provide important context, as the same workload may behave differently depending on the device on which it is executed.

To make the workload composition usable during analysis, the module encodes the active workloads as a binary vector. Each supported workload is represented by a separate feature, where a value of 1 indicates that the workload is present in the experiment, and a value of 0 indicates that it is not. This allows co-located workload combinations to be represented in a consistent numerical format.

Finally, the module summarizes the utilization metrics collected during execution. For each resource category, namely input/output, memory, network, CPU, and power, three statistical values are computed: peak, mean, and fluctuation. The peak value represents the maximum observed value of a resource metric during the experiment, while the mean value represents the average value across the monitoring period. The fluctuation value captures the variability of the metric over time and is calculated using the coefficient of variation, which is the standard deviation divided by the mean.

The resulting fingerprint structure is shown in Figure 5, where the main feature groups are organized into application context, target performance value, hardware specifications, workload identity flags, and resource utilization summaries. Together, these components produce a compact but informative representation of each experiment, combining application context, hardware characteristics, workload identity, resource behavior, and performance into a single fingerprint record.

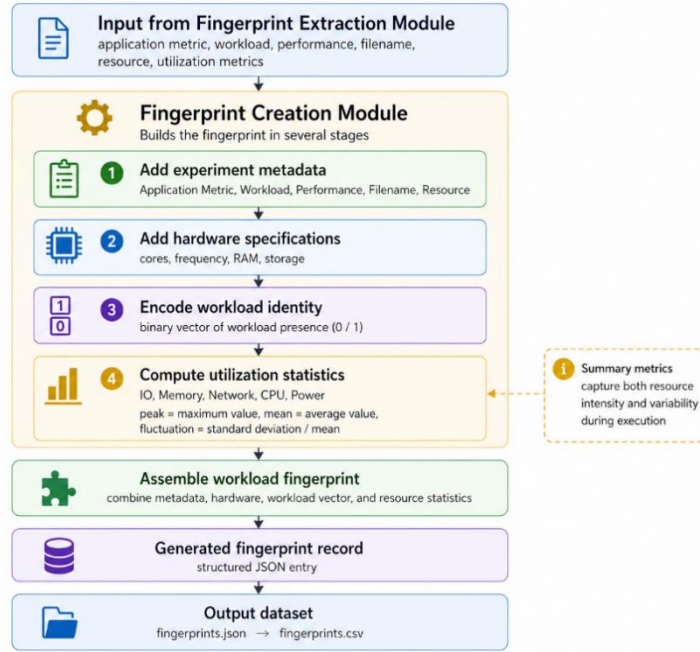


Figure 7. Fingerprint Creation Module

### 3.1.3 Dataset Construction

After the individual workload fingerprints are generated, they are collected into a single JSON file named fingerprints.json. This file stores each fingerprint as a structured record containing the application metric, workload identity, hardware characteristics, performance value, and summarized utilization features. The JSON format is particularly useful at this stage because new fingerprints can be added incrementally as additional experiments are processed.

As illustrated in Figure 8, the dataset construction process begins by loading all fingerprint records from the JSON file and validating that the file contains a list of structured entries. Once the records are loaded, the pipeline performs an additional preprocessing step on the fluctuation features. For each resource, namely input/output, memory, CPU, network, and power, the fluctuation values are grouped per device and classified into three categories: low, mid, and high. These categories are defined using the 33rd and 66th percentiles of the fluctuation values for each device. This allows the

pipeline to represent resource variability in a relative way, considering the behaviour of each hardware resource separately.

The records are then converted into a tabular data frame. During this step, the pipeline also derives an additional hardware feature called `cpu_capacity`, calculated by multiplying the number of cores by the processor frequency. This provides a compact representation of the computational capacity of each device and can be used as an additional input feature during analysis.

After the full data frame is constructed, the dataset is separated according to the application metric. This step is necessary because different workloads use different performance targets. For example, database performance, machine learning inference performance, streaming latency, and streaming tuples represent different types of output values and should therefore be analyzed separately. As a result, the pipeline produces four application-specific datasets, covering database workloads, machine learning workloads, streaming latency, and streaming tuples.

These CSV files form the final structured datasets used in the next stages of the thesis. They provide the input for data preparation, regression-based performance prediction, feature importance analysis, grouped resource analysis, and clustering. In this way, the dataset construction stage transforms the raw fingerprint records into organized machine-learning-ready datasets.

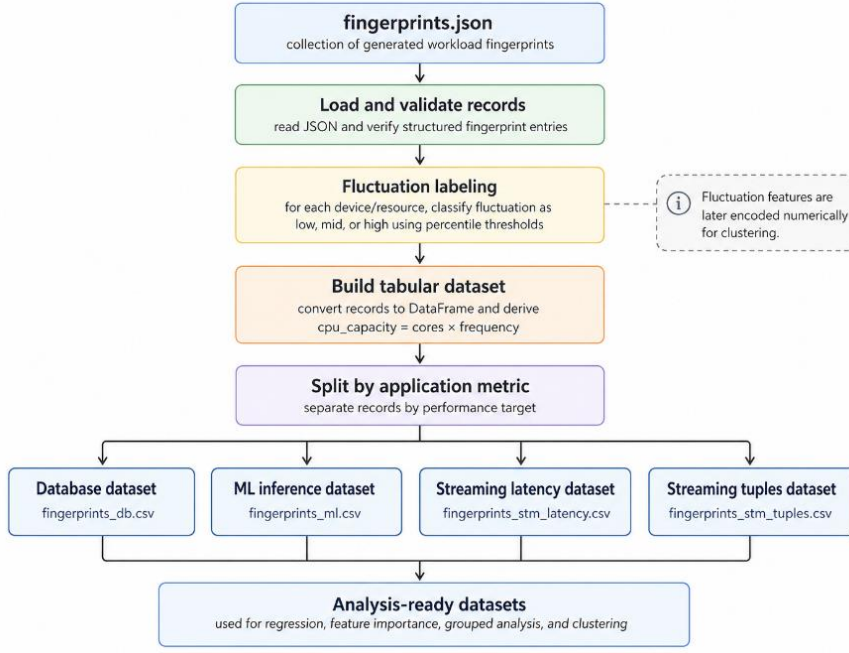


Figure 8. Dataset construction process

### 3.2 Data Preparation and Performance Modeling

As described in the previous section, the fingerprint dataset is separated according to the application metric, producing distinct datasets for database performance, machine learning inference, streaming latency, and streaming tuples. This separation is necessary because each application metric represents a different type of performance target. For example, database performance is measured differently from machine learning inference or streaming latency. Therefore, combining all records into a single training dataset would mix different performance meanings and could lead to misleading model behavior.

For this reason, data preparation, training, and evaluation are performed independently for each application-specific dataset. This ensures that each regression model learns relationships between workload characteristics, resource utilization, hardware features, and performance values that are consistent within the same application category. The following subsections describe the preprocessing steps, the regression models used, and the evaluation procedure applied to each dataset.

### 3.2.1 Data Preparation

For each application-specific dataset, the data preparation process follows the pipeline shown in Figure 9. It begins by separating the target variable from the input features. The target column is Performance, which represents the observed application-level performance value that the regression models aim to predict.

Before training, several columns are removed from the feature set, namely Workload, Filename, Resource, storage, and Application Metric. These columns are excluded because they mainly act as identifiers or metadata rather than direct predictive features. For example, Workload and Filename describe the experiment record, while Application Metric is already used to split the dataset into separate performance categories. The Resource column is also removed because it only identifies the device name. The characteristics of each device are already represented through hardware-related features, such as the number of cores, processor frequency, and RAM capacity, making the resource name itself unnecessary for prediction. The storage field is removed because it did not provide useful variation for the prediction task.

After these columns are removed, the remaining features are used as input variables. The dataset is then split into training and testing subsets, with 80% of the data used for training and 20% used for testing. This split allows the models to learn from one part of the data and then be evaluated on unseen records, providing a more realistic estimate of their predictive performance.

Any categorical features that remain in the dataset are converted into numerical form using ordinal encoding. This step is necessary because the regression models require numerical input values. The encoder is fitted only on the training data and then applied to the test data, while unknown categories are handled using a predefined value. Finally, all feature columns are converted to numeric form to ensure compatibility with all models, including XGBoost.

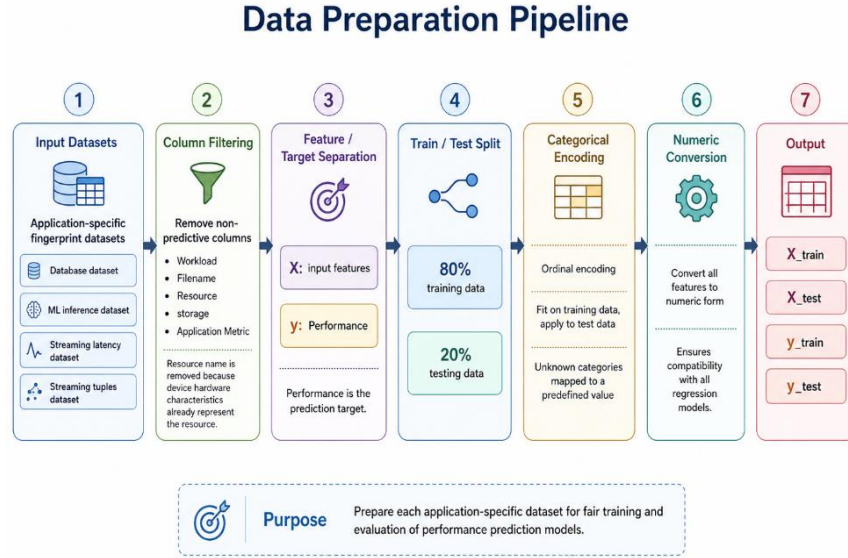


Figure 9. Dataset Preparation Pipeline

### 3.2.2 Regression Model Selection

After preprocessing, each application-specific dataset is used to train and evaluate a set of regression models. The models include Linear Regression [17], Decision Tree Regressor [18], Random Forest Regressor [19], Extra Trees Regressor [20], Gradient Boosting Regressor [21], HistGradientBoosting Regressor [22], AdaBoost Regressor [23], and XGBoost Regressor [24]. These models were selected to compare different levels of modelling complexity, ranging from simple linear approaches to more flexible tree-based and ensemble-based methods.

Linear Regression is used as a simple and interpretable baseline, allowing an initial assessment of whether workload performance can be approximated through linear relationships between the fingerprint features and the target performance value [17]. Decision Tree Regression is included because it can capture non-linear relationships and feature interactions without requiring strong assumptions about the distribution of the data [18]. However, a single decision tree may be sensitive to the training data and prone to overfitting, which motivates the use of ensemble methods.

Random Forest and Extra Trees aggregate multiple decision trees to improve robustness and reduce variance [19], [20]. These models are relevant in this context because workload performance may depend on interactions between several resource dimensions, such as CPU, memory, input/output, network usage, and power consumption, rather than on a single dominant feature. In addition, Gradient Boosting, HistGradientBoosting, AdaBoost, and XGBoost are included as representative boosting approaches. These methods build ensembles iteratively, where each new learner aims to improve the errors made by previous learners [21]–[24].

Tree-based ensemble methods are particularly suitable for structured tabular datasets, and recent studies show that they often remain highly competitive for tabular prediction tasks [26], [27]. By comparing these models, the analysis evaluates whether simple linear models are sufficient or whether more flexible tree-based and ensemble-based approaches are required to model workload performance in multi-tenant edge environments.

### **3.2.3 Model Evaluation Procedure**

After training the regression models, each model is evaluated on the corresponding test set of each application-specific dataset. The evaluation is performed separately for the database, machine learning inference, streaming latency, and streaming tuples datasets, since each dataset represents a different performance target.

The main evaluation metrics used are the coefficient of determination,  $R^2$  [28], and Mean Absolute Error, MAE [29]. The  $R^2$  score indicates how much of the variation in the observed performance values is explained by the model. A higher  $R^2$  value suggests that the model captures the relationship between the fingerprint features and the target performance value more effectively [28]. MAE measures the average absolute difference between the predicted and observed performance values, providing an interpretable measure of prediction error [29].

In addition to the absolute MAE value, the prediction error is also expressed as a percentage of the mean target value. This is calculated by dividing the MAE by the mean of the observed performance values in the test set and multiplying the result by 100.

Reporting this percentage makes the prediction error easier to compare across datasets with different performance scales.

For each dataset, all regression models are trained and evaluated using the same train-test split. The results are then returned in descending order based on the test  $R^2$  score, so that the best-performing models appear first. The model with the highest test  $R^2$  is considered the best-performing model for that dataset and is used in the following analysis stages, including feature importance and grouped resource impact analysis. The actual evaluation results are presented and discussed in Chapter 4.

### **3.3 Feature Importance**

To better understand which factors influence workload performance, feature importance analysis is applied to the trained regression models for each application-specific dataset. While the regression results indicate how accurately each model predicts performance, they do not directly explain which fingerprint features contribute most to the prediction. Feature importance analysis addresses this by identifying the features that have the strongest impact on the model output.

In this thesis, feature importance is examined at two different levels. First, individual feature importance is calculated to evaluate the contribution of each feature separately, such as CPU mean usage, memory peak usage, power fluctuation, or workload identity indicators. Second, grouped feature importance is applied by organizing related features into higher-level categories, including Power, CPU, Memory, Input/Output, Network, and Workload Identity. This grouped analysis provides a broader view of which resource dimensions contribute most to performance variation.

By combining individual and grouped feature importance, the analysis provides both a detailed and a high-level understanding of workload behavior. This makes it possible to identify not only specific influential features, but also the resource categories that are most relevant for predicting performance in multi-tenant edge environments.

### 3.3.1 Individual Feature Importance

Individual feature importance is used to examine the contribution of each fingerprint feature separately. The purpose of this analysis is to identify which specific features, such as CPU usage, memory usage, network activity, power consumption, or workload identity indicators, are most influential when predicting workload performance. Since each application metric is modelled independently, the analysis is also performed separately for each application-specific dataset.

For each dataset, feature importance is applied to the regression model that achieved the highest test  $R^2$  score during model evaluation. This allows the interpretation step to focus on the model that best captures the relationship between the fingerprint features and the corresponding performance target.

The first method used is permutation feature importance [30]. This method evaluates the importance of each feature by measuring how much the model performance decreases when the values of that feature are randomly shuffled. If shuffling a feature causes a large drop in the  $R^2$  score, this indicates that the model relies strongly on that feature for prediction. In contrast, if shuffling a feature has little effect on model performance, the feature is considered less important. In this implementation, permutation importance is computed on the test set using  $R^2$  as the scoring metric, and the process is repeated multiple times to obtain a more stable estimate. The implementation follows the permutation importance functionality provided by scikit-learn [22].

The resulting importance values are sorted in descending order, so that the most influential features appear first. The top features are then visualized using a horizontal bar chart, where the x-axis represents the decrease in  $R^2$  caused by permuting each feature.

In addition to permutation importance, SHAP-based analysis is applied to provide a more detailed interpretation of the selected model [16]. While permutation importance provides a global view of how much each feature affects model performance, SHAP explains how individual features contribute to the model predictions. For tree-based models, a SHAP TreeExplainer is used, with a sample of the training data serving as the background

distribution. The generated SHAP summary plots show both the overall importance of each feature and the direction of its effect on the predictions [16].

By combining permutation importance and SHAP analysis, the individual feature importance stage provides a more complete view of how each fingerprint feature influences performance prediction. The numerical results and visualizations produced by this analysis are presented and discussed in Chapter 4.

### **3.3.2 Grouped Feature Importance**

After examining individual features, the analysis is extended to feature groups to provide a higher-level view of performance influence. Instead of assessing each fingerprint feature separately, grouped feature importance evaluates broader resource categories and helps identify whether performance prediction is mainly driven by resource utilization, hardware characteristics, or workload identity.

As shown in Figure 10, the fingerprint features are organized into six groups: Power, CPU, Input/Output, Memory, Network, and Workload Identity. Each group contains the corresponding summary statistics or indicators associated with that category. For example, the CPU group includes CPU mean, peak, fluctuation, capacity, number of cores, and processor frequency, while the Workload Identity group contains the binary workload indicators.

Before applying the grouped analysis, the available feature columns are matched with their predefined feature groups. This ensures that only features present in the current application-specific dataset are included in the calculation.

For grouped permutation importance, all features belonging to the same group are shuffled together on the test set. The decrease in  $R^2$  from the baseline model performance is then used as the importance value of that group. In this way, the method estimates how much the model depends on an entire resource category rather than on a single feature.

Grouped SHAP analysis is also used to estimate group-level influence. After SHAP values are computed for the selected model, the absolute SHAP values of all features belonging to the same group are summed for each test sample. These summed values are then averaged across all test samples to produce a single SHAP-based importance score for each feature group.

The grouped importance scores are sorted and visualized using horizontal bar charts, where longer bars indicate stronger group-level influence on the model prediction. The results of the grouped feature importance analysis are presented and discussed in Chapter 4.

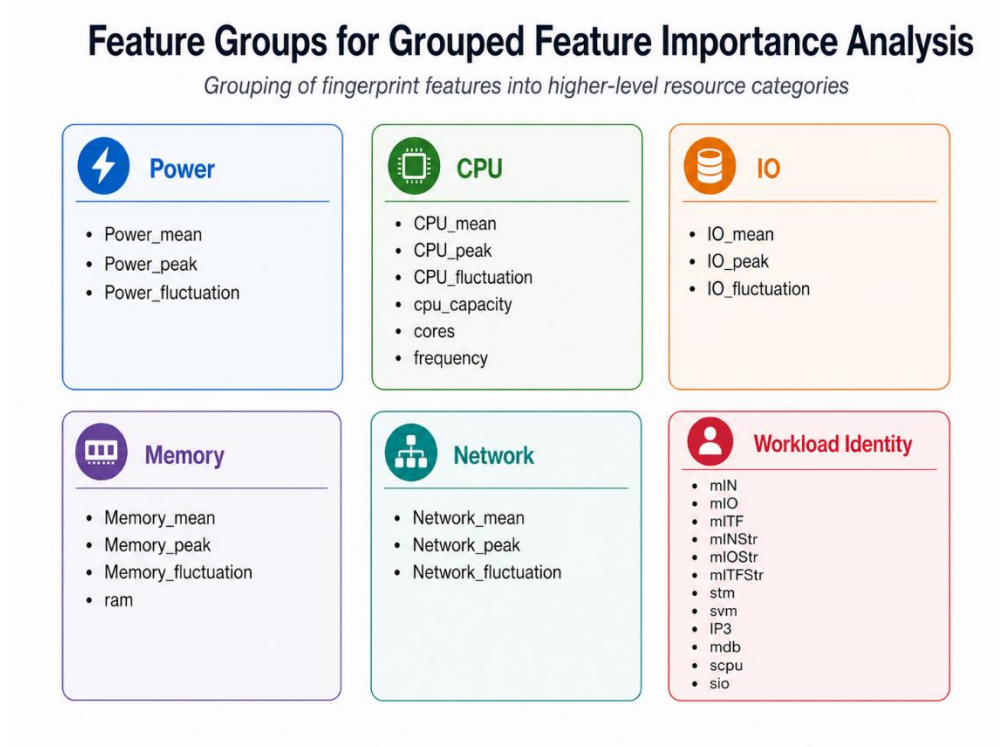


Figure 10. Feature Groups

### 3.4 Clustering

While feature importance analysis helps identify which features have the strongest influence on performance prediction, clustering is used to explore similarities between workload executions based on their resource utilization patterns. The purpose of this

analysis is not to predict performance directly, but to group fingerprints with similar behaviour and provide additional insight into workload characterization.

The clustering analysis is performed separately for each application-specific dataset. Before applying clustering, columns that do not describe runtime resource behaviour are removed from the feature matrix. These include the performance target, workload identity indicators, hardware-related features, and metadata fields such as workload name, filename, resource name, and application metric. This is done because the goal of clustering is to group experiments according to their observed resource utilization patterns, rather than grouping them based on workload name, device type, or performance value. As a result, the clustering process focuses mainly on features such as CPU, memory, input/output, network, and power usage.

Since the clustering features have different numerical scales, the data is standardized before applying the clustering algorithm. Standardization ensures that features with larger numerical ranges do not dominate the clustering process. After scaling, the KMeans algorithm is applied to group similar fingerprint records into clusters [25].

To support the selection of the number of clusters, the elbow method and silhouette score are used. The elbow method examines how inertia changes for different values of  $k$ , where inertia measures how compact the clusters are. The silhouette score provides an additional indication of how well-separated the clusters are. By analysing both plots, an appropriate number of clusters is selected for each application-specific dataset.

After choosing the number of clusters, KMeans is fitted to the scaled feature matrix. The resulting cluster centroids are then transformed back to the original feature scale to make them easier to interpret. The centroid matrix summarizes the average resource utilization pattern of each cluster and helps characterize the different types of workload behaviour observed in the dataset.

In addition to the numerical centroid matrices, normalized heatmaps are generated to visualize the resource profile of each cluster. For each application-specific dataset, the centroid values are normalized feature-wise so that differences across resources with

different scales can be compared more easily. In these heatmaps, each row represents a cluster, and each column represents a resource utilization feature, while the colour intensity indicates the relative magnitude of that feature within the dataset. This visualization makes it easier to identify patterns such as CPU-intensive, input/output-intensive, network-heavy, memory-intensive, or high-power clusters.

The clustering results, centroid matrices, and normalized centroid heatmaps are presented and discussed in Chapter 4.

### **3.5 BenchPilot Interface Design and Implementation**

This section presents the design and implementation of the BenchPilot interface developed as part of this thesis. The interface extends the original BenchPilot workflow by providing a unified web-based environment for experiment configuration, deployment, monitoring, and result management. Instead of requiring users to manually edit configuration files, execute terminal commands, inspect logs, and retrieve results from separate locations, the interface integrates these steps into a single system.

The purpose of this implementation is to reduce manual effort, improve usability, and support more scalable and reproducible experimentation in multi-tenant edge environments. The following subsections describe the main design goals, system architecture, user management, experiment deployment workflow, monitoring features, result access mechanisms, and integration with CSV output and Jupyter-based analysis.

#### **3.5.1 Design Goals**

As discussed in Section 2.3.4, the original BenchPilot workflow supports repeatable benchmarking, but several stages of the experimentation process still require manual work and are split across separate tools and steps. During the data collection phase of this thesis, these limitations became particularly evident, since experiments had to be configured, deployed, monitored, and analysed across multiple workloads, devices, and co-location scenarios. This required users to edit YAML files manually, execute deployment commands through the terminal, inspect Docker services separately, locate

output files, and switch between tools such as Jupyter, the file system, and command-line utilities.

The BenchPilot interface was designed to address these limitations by providing a unified environment for managing the complete experiment lifecycle. Its design goals focus on reducing manual effort, improving reproducibility, supporting multi-user experimentation, simplifying infrastructure and configuration management, and enabling easier access to logs, results, and post-experiment analysis. These goals guided the design and implementation of the system and define the main capabilities provided by the interface.

As shown in Figure 11, the design goals are organized into six categories: workflow automation and lifecycle integration, reproducibility and experiment organization, multi-user and administrative support, infrastructure and configuration management, monitoring and deployment safety, and result analysis and validation. This organization reflects the main limitations identified in the original BenchPilot workflow and provides the basis for the interface design described in the following subsections.

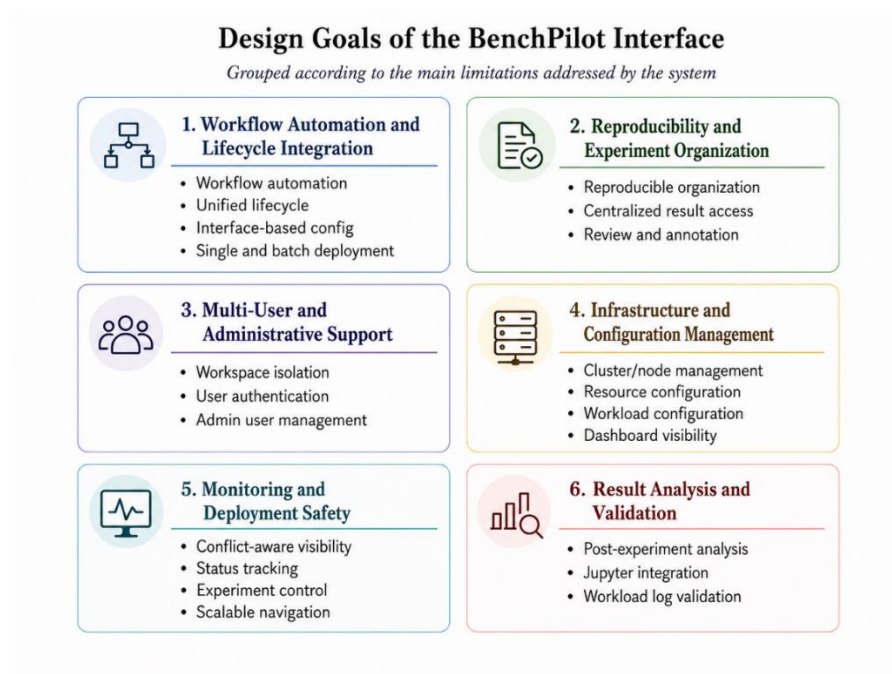


Figure 11. Design goals of the BenchPilot Interface

### 3.5.1.1 Workflow Automation and Lifecycle Integration

The first design goal of the BenchPilot interface was to automate the main steps of the experimentation workflow and integrate them into a single lifecycle. In the original BenchPilot workflow, users had to manually create or edit YAML configuration files, define output paths, execute deployment commands through the terminal, and later locate the generated logs and CSV files. Although this process is manageable for a small number of experiments, it becomes repetitive and error-prone when many workload combinations, devices, and co-location scenarios must be explored.

To address this limitation, the interface supports experiment creation, configuration generation, deployment, monitoring, and result access from a single web-based environment. Instead of writing YAML files manually, users define experiments by selecting workloads, nodes, and deployment options through the interface. The system then generates the corresponding BenchPilot-compatible configuration files automatically, allowing experiments to move directly from configuration to deployment within the same workflow.

The deployment process is also integrated into the interface. Once an experiment has been configured, it can be launched without requiring the user to execute terminal commands manually. The same approach applies to batch execution, where multiple experiments can be defined and submitted together. This is particularly useful for large experimental campaigns, where many workload combinations must be executed sequentially and consistently.

Overall, this design goal transforms the BenchPilot workflow from a set of disconnected manual actions into a continuous experiment lifecycle. By automating configuration generation and deployment, and by connecting these steps with monitoring and result access, the interface reduces manual effort, lowers the risk of configuration mistakes, and makes large-scale experimentation easier to manage.

### 3.5.1.2 Reproducibility and Experiment Organization

A second design goal of the BenchPilot interface was to improve the reproducibility and organization of experiments through both structured file storage and persistent database-backed metadata. In the original workflow, experiment files and outputs were distributed across manually defined locations, making consistent organization difficult as the number of runs increased. This also made it harder to trace each result back to the configuration that produced it.

To address this issue, the interface uses a user-specific workspace structure. Each user has a separate directory where generated YAML files, experiment outputs, logs, CSV files, and notebooks are stored. This prevents results from different users from being mixed and makes it easier to locate the files associated with a specific experiment. In addition, the interface records key metadata for each run, including the experiment name, status, timestamp, process identifier, generated YAML file, execution directory, log file, JSON output, CSV output, and notebook location when applicable.

In addition to the file-based workspace organization, the interface uses MongoDB to store and manage experiment metadata. MongoDB is used to persist information about users, experiments, batches, execution status, timestamps, process identifiers, comments, review states, and file locations. This database layer allows the interface to retrieve experiment history, display previous runs, update experiment status, and associate generated outputs with the correct user and experiment. By storing this metadata independently from the raw output files, the system provides a more reliable way to track experiment state and maintain the connection between configuration, execution, and results.

This structure supports reproducibility by preserving both the configuration and the outputs of each experiment. Completed runs can therefore be inspected, repeated, or compared more easily. Users do not have to rely on memory or manual notes to determine which configuration produced a result, because the interface maintains the connection between the experiment definition, execution state, and generated artifacts.

The interface also allows users to review completed experiments and add comments. Review states can be used to mark whether a completed experiment should be considered successful or failed after manual inspection, while comments provide additional context about unusual behaviour, errors, or observations. This is useful during large experimental campaigns, where many runs must later be filtered, validated, or used for dataset construction.

Overall, this design goal ensures that experiments are treated as structured records rather than isolated executions. By consistently linking configurations, database metadata, logs, results, and analysis artifacts, the interface improves traceability, supports reproducible experimentation, and simplifies the management of large collections of BenchPilot runs.

### **3.5.1.3 Multi-user and Administrative Support**

Another important design goal of the BenchPilot interface was to support multi-user experimentation. In the original workflow, experiments were mainly managed through shared files, terminal commands, and common directories. When multiple users interact with the same BenchPilot installation or edge infrastructure, this can make it difficult to determine who created a particular experiment, where its outputs are stored, or which workloads are currently running.

To address this issue, the interface includes user authentication and user-specific workspaces. Each user accesses the system through a separate account, and user accounts are stored in MongoDB together with the information required to associate experiments, batches, and generated artifacts with the correct user. The experiments created through the interface are therefore linked to the user who created them. Generated YAML files, batch outputs, CSV files, logs, and notebooks are stored under user-specific directories. This separation reduces confusion between users experiments and makes it easier to trace who created, deployed, or analysed a particular run.

The interface also includes administrative functionality. Administrator users can manage registered users through the interface, including adding or removing accounts when

needed. This provides controlled access to the experimentation environment and avoids relying on manual account or file management outside the system. By separating regular user functionality from administrative actions, the interface supports safer shared use of the platform.

Multi-user support is particularly important in shared edge experimentation environments, where several users may need to conduct experiments on the same pool of devices. By associating experiments, files, and metadata with specific users, the system improves accountability and organization. At the same time, administrative controls make it easier to manage access to the platform as the number of users grows.

Overall, this design goal ensures that the BenchPilot interface can operate as a shared experimentation environment rather than a single-user tool. User authentication, workspace isolation, database-backed ownership tracking, and administrative management help organize experiments more clearly, reduce the risk of file or result confusion, and support collaborative use of the edge benchmarking infrastructure.

#### **3.5.1.4 Infrastructure and Configuration Management**

Another design goal of the BenchPilot interface was to simplify the management of the experimental infrastructure and its configuration. In the original BenchPilot workflow, changes to the cluster, resource descriptions, or workload settings often required users to manually edit configuration files. This process is error-prone, especially when multiple devices, workload templates, and runtime parameters must be maintained consistently across experiments.

To address this limitation, the interface provides web-based functionality for managing the BenchPilot cluster. Users can inspect the available nodes and view their relevant information, while authorized users can add, edit, remove, or reset nodes when needed. This reduces the need to modify cluster configuration files directly and makes the current infrastructure state easier to understand before deployment.

The interface also supports resource configuration management. BenchPilot experiments depend on resource descriptions such as node names, hardware characteristics, mounts, CPU cores, memory, and other specifications. By allowing these values to be viewed and managed in a structured way, the interface helps ensure that generated experiments use consistent and up-to-date resource information.

In addition, workload templates and runtime settings can be inspected and modified through the interface. This allows users to adjust experiment behaviour without manually editing internal template files. This is important because different workloads may require different parameters, such as server IP addresses, ports, thread counts, injection rates, or workload-specific configuration options.

Overall, this design goal extends the interface beyond experiment execution by also supporting the configuration of the infrastructure and workload settings required for reliable experimentation. By integrating cluster, resource, and workload management into the same environment used for deployment, the interface improves usability, reduces manual configuration effort, and supports more consistent experiment generation across different devices and workload combinations.

### **3.5.1.5 Monitoring and Deployment Safety**

Another design goal of the BenchPilot interface was to improve monitoring and deployment safety during experiment execution. In the original workflow, users had to inspect terminal output, execution logs, and Docker services manually to understand whether an experiment was running correctly. This made it difficult to maintain a clear overview of the system state, especially when multiple experiments or users were involved.

To address this limitation, the interface provides direct visibility into experiment execution status. Experiments are tracked using states such as pending, running, finished, failed, killed, and reviewed. These states allow users to understand the progress and outcome of each experiment without manually checking output files. The interface also

displays running experiments and Docker services, giving users a clearer view of what is currently active in the system.

This visibility is important for deployment safety. In a shared edge environment, launching a new experiment without knowing what is already running may lead to workload conflicts or resource contention. By showing active experiments and Docker services, the interface helps users avoid accidental overlapping deployments and makes it easier to identify when a node or service is already in use.

The interface also provides experiment control actions. Users can stop running experiments or batches directly from the interface when a deployment fails, hangs, or needs to be cancelled. This avoids the need to manually locate processes or issue terminal commands to terminate execution, making recovery from failed or unwanted runs faster and safer.

In addition, the experiment view supports filtering, sorting, pagination, and search functionality. These features are important for large experimental campaigns, where many experiments may be executed over time. Users can locate specific experiments, inspect their status, and access relevant logs more efficiently, rather than navigating manually through directories or long lists of files.

Overall, this design goal makes experiment execution more transparent and easier to control. By providing status tracking, running experiment visibility, Docker service inspection, stop actions, and scalable navigation, the interface reduces the risk of conflicting deployments and improves experiment management during execution.

#### **3.5.1.6 Result analysis and validation**

The final design goal of the BenchPilot interface was to simplify result analysis and support early validation of experiment outputs. In the original workflow, users had to manually locate output files, inspect logs, export CSV results, and create analysis notebooks. This required switching between the file system, terminal, Jupyter, and custom scripts, increasing the effort required after each experiment completed.

To address this limitation, the interface provides centralized access to experiment outputs. For each completed experiment, users can inspect the generated JSON log, CSV output, YAML file, execution logs, and error information directly from the experiment view. This reduces manual file navigation and allows users to quickly check whether the experiment produced the expected artifacts.

The interface also supports CSV result inspection. Completed experiment CSV files can be previewed in the browser and downloaded when needed. This allows users to verify the structure and content of the exported metrics before using them in later analysis. Since the fingerprint extraction and performance modelling pipeline depends on consistent CSV outputs, direct access to these files helps reduce the risk of using incomplete or incorrect results.

To further support post-experiment analysis, the interface integrates with Jupyter notebooks. When an experiment has completed and a CSV output exists, the user can open the CSV in a generated Jupyter notebook directly from the interface. The notebook loads the experiment CSV into a pandas DataFrame, allowing the user to continue with exploratory analysis, visualization, or debugging without manually creating a notebook or locating the CSV path.

In addition to result access, the interface includes workload-specific validation notes. These notes provide quick feedback about whether the main expected outputs of an experiment appear to have been produced correctly. They do not replace full manual analysis, but they help users identify obvious failures, incomplete runs, or suspicious outputs more quickly before using the results in later processing.

Overall, this design goal reduces the effort required after experiment execution. By centralizing logs and outputs, supporting CSV preview and download, integrating with Jupyter notebooks, and providing automated workload validation notes, the interface helps users move more efficiently from completed experiments to analysis-ready data.

### 3.5.2 System Architecture

The BenchPilot interface follows a web-based client-server architecture that extends the existing BenchPilot framework without replacing its core execution mechanisms. The system consists of a React frontend, a FastAPI backend, a MongoDB metadata store, a user-specific file workspace, the existing BenchPilot execution layer, the edge cluster infrastructure, and a Jupyter notebook environment for post-experiment analysis. React was selected for the frontend because it supports the development of reusable user interface components [33]. FastAPI was selected for the backend because it is a Python web framework for building APIs based on standard Python type hints [34]. MongoDB was used as the database layer because it stores data as BSON documents and supports a flexible document-based data model [35], [36]. Figure 12 presents the overall architecture of the system.

The React frontend provides the user-facing environment through which users configure experiments, manage resources, deploy workloads, monitor execution, inspect logs, access results, and open generated notebooks. It communicates with the backend through HTTP API requests and presents the current system state, including experiment status, running deployments, Docker services, stored results, and available output artifacts.

The FastAPI backend acts as the central coordination layer of the system. It is responsible for authentication, user management, cluster and resource management, workload configuration, YAML generation, experiment deployment, batch execution, status tracking, result retrieval, notebook creation, and workload-specific validation. When a user deploys an experiment, the backend generates the required BenchPilot-compatible YAML file, creates the corresponding run directories, stores metadata in MongoDB, and launches the appropriate BenchPilot execution process.

MongoDB is used to store persistent metadata for the interface. This includes user accounts, user-specific workload configurations, experiment records, batch records, execution status, timestamps, process identifiers, comments, review states, and output paths. In addition, when an experiment completes, selected output contents are copied

into MongoDB, including the JSON log payload, CSV file contents, and execution log text, so that they can be displayed directly through the interface.

In addition to the metadata store, the system maintains a file-based workspace under the BenchPilot interface directory. Each user has a separate workspace in which generated YAML files, batch files, experiment outputs, CSV results, execution logs, and Jupyter notebooks are stored. While MongoDB stores metadata, selected output contents, and file paths, the actual experiment artifacts remain organized in the corresponding user workspace.

The backend interacts with the existing BenchPilot execution layer to run experiments. Once a configuration is generated, the backend launches the BenchPilot deployment process, which deploys the selected workloads as Docker services on the target edge nodes. During execution, the system monitors the process status and updates the corresponding MongoDB records. After an experiment completes, the generated JSON logs, CSV outputs, and execution logs become accessible through the interface.

The interface also integrates with Jupyter to support post-experiment analysis. When a completed experiment contains a CSV output, the backend can create a notebook in the user workspace and return a corresponding Jupyter URL. The frontend then opens this notebook directly in the running Jupyter server, allowing users to continue analysis without manually locating the CSV file or writing the initial notebook setup code.

Overall, this architecture connects the existing BenchPilot benchmarking workflow with a web-based management layer. The frontend improves usability, the backend coordinates experiment lifecycle operations, MongoDB stores persistent metadata and selected output contents, the user workspace stores generated artifacts, and the existing BenchPilot execution layer continues to perform the actual deployment and execution of workloads on the edge infrastructure.

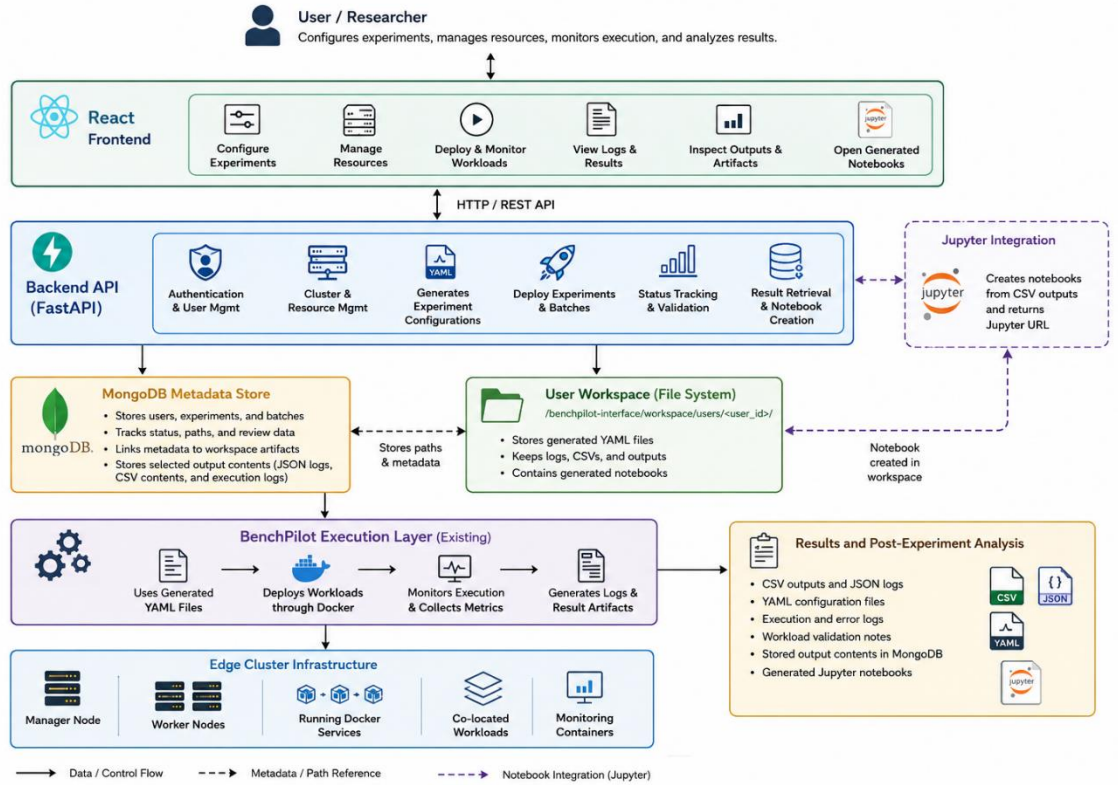


Figure 12. System architecture of the BenchPilot interface

### 3.5.3 Backend Implementation

This section describes the backend implementation of the BenchPilot interface. The backend is responsible for coordinating user authentication, configuration management, experiment deployment, execution monitoring, result storage, and integration with external tools such as Docker and Jupyter.

#### 3.5.3.1 Backend Role and Application Startup

The backend of the BenchPilot interface was implemented using FastAPI and acts as the coordination layer between the web interface, MongoDB, the user file workspace, the existing BenchPilot execution scripts, Docker services, and the Jupyter notebook environment. Its main responsibility is to expose the operations required by the frontend through HTTP API endpoints, while preserving the existing BenchPilot workflow for experiment execution. In this way, the backend does not replace BenchPilot's core deployment and metric collection mechanisms, but wraps them with additional

functionality for authentication, configuration management, experiment tracking, result retrieval, and post-experiment analysis.

During application startup, the backend initializes a connection to MongoDB using the configuration values defined in the environment file. The database connection is stored in the application state so that it can be accessed by the different API routes. The backend also configures Cross-Origin Resource Sharing to allow communication with the React frontend and registers the main routers used by the interface, including authentication, user management, cluster management, workload configuration, experiment deployment, batch execution, Docker service inspection, and user profile retrieval. A health endpoint is also provided to verify that the backend can communicate with MongoDB.

### **3.5.3.2 Authentication and User Management**

Authentication is handled through session-based access. When a user logs in, the backend verifies the submitted cred

entials against the stored user account in MongoDB. If authentication succeeds, a session token is generated and stored in the `auth_sessions` collection together with its expiration time. The token is then returned to the browser as an HTTP-only cookie. Protected backend routes use a shared authentication dependency that checks the session token, retrieves the corresponding user from MongoDB, and removes sensitive fields such as passwords and authorization keys before returning the user object. The backend also supports authorization keys for internal batch execution, allowing batch runner processes to invoke protected deployment endpoints without requiring an interactive browser session.

User management is implemented through dedicated backend endpoints. New users can be created either through sign-up or by an administrator. When a user is created, the backend stores the account in the `users` collection, hashes the password, creates an authorization key, initializes default workload templates and runtime configuration, and creates a user-specific workspace directory under the interface's `users` directory. Administrator users can also retrieve the list of registered users and delete accounts. This

supports controlled access to the interface and links each experiment, batch, and generated artifact to a specific user.

### **3.5.3.3 Cluster and Workload Configuration**

The backend also manages cluster and workload configuration. Cluster endpoints allow users to retrieve the available nodes, while authorized users can add, edit, remove, or reset nodes. Node information is stored per user in MongoDB, allowing each user to maintain resource-specific settings while administrator-level actions can still update the shared cluster definition.

Workload configuration endpoints allow users to inspect, edit, and reset workload templates and runtime parameters, such as experiment templates and workload-specific values. These configuration values are stored in each user's MongoDB record and are later used during YAML generation.

### **3.5.3.4 Experiment Deployment and YAML Generation**

Experiment deployment is one of the main backend responsibilities. When a user deploys an experiment, the backend first resolves the current user and creates a structured run directory inside the user workspace. This directory contains the generated YAML file, result folders, CSV output directory, metadata JSON path, and execution log file.

The backend then generates a BenchPilot-compatible YAML configuration using the selected experiment name, the user's resource specification, workload templates, and runtime configuration. Before launching the experiment, an experiment record is inserted into MongoDB with status pending, together with metadata such as the user ID, experiment name, timestamp, run directory, YAML file, metadata JSON path, CSV directory, and log file path.

### **3.5.3.5 Conflict Handling and Process Launching**

Before execution begins, the backend checks for deployment conflicts by comparing the requested resource with currently running experiments stored in the `running_experiments` collection. If another experiment is already running on the same resource, the new experiment is marked as failed and the deployment is rejected. This conflict check helps reduce accidental overlapping deployments on the same edge node.

If no conflict is detected, the backend launches the BenchPilot runner script as a background process using the generated YAML file and output paths. The process identifier is stored in MongoDB, the experiment status is updated to running, and the record is synchronized with the `running_experiments` collection.

### **3.5.3.6 Experiment Monitoring and Result Storage**

Experiment monitoring is handled asynchronously by a backend monitoring service. The monitor checks whether the experiment process is still alive, inspects execution logs, detects success and failure patterns, and verifies whether expected outputs such as the metadata JSON file and CSV files have been produced.

When an experiment finishes, the monitoring service updates the corresponding MongoDB record with the final status. If available, selected completed outputs are also loaded into MongoDB, including the JSON log payload, CSV file contents, and execution log text. This allows the frontend to display logs and CSV results directly without requiring the user to manually open files from the workspace. If the experiment fails, the monitor records an error summary and removes the experiment from the `running_experiments` collection.

### **3.5.3.7 Batch Execution**

Batch execution is implemented through a separate backend route and runner script. When a batch is submitted, the backend creates a dedicated batch run directory in the user workspace to store the run metadata, execution state, deployment tracking information, and logs.

A batch record is then inserted into MongoDB with the batch status, timestamp, run directory, log file path, and list of experiment names. The batch runner executes experiments sequentially, updates the batch record as each experiment starts and finishes, and records the number of successful and failed executions. The backend also provides endpoints for listing batches, retrieving batch status, and stopping a running batch.

### **3.5.3.8 Experiment Control and Docker Service Inspection**

The backend provides several routes for retrieving and controlling experiments after deployment. Users can list their experiments, retrieve the full status of an experiment, access live or stored execution logs, update review status, add comments, and stop running experiments. Stopping an experiment updates its status, records the failure reason as a user action, removes it from the running collection, and attempts to terminate the corresponding process and clean up services associated with the experiment. Similar control logic is provided for batches, allowing users to cancel long running or problematic batch executions from the interface.

Docker service visibility is provided through backend endpoints that execute Docker commands on the server. The backend retrieves the list of running Docker services and can also inspect the task list of a specific service. The command output is parsed into structured response objects before being returned to the frontend. This allows the interface to show Docker-level state without requiring users to manually run Docker commands in the terminal.

### **3.5.3.9 Jupyter Notebook Integration**

Finally, the backend implements Jupyter notebook integration for completed experiments. When a user requests to open a CSV in Jupyter, the backend verifies that the experiment belongs to the current user, checks that the experiment has finished, locates the CSV file from the experiment's stored CSV directory, and creates a notebook in the user workspace.

The notebook path mirrors the experiment's run directory under the user's notebooks folder and is created only once for each experiment. If the notebook already exists, the backend returns the existing notebook URL instead of recreating the file. The generated notebook contains a short markdown summary and Python code that loads the CSV into a pandas DataFrame. The backend then returns a Jupyter URL, which the frontend opens directly in the running Jupyter server.

### **3.5.3.10 Summary**

Overall, the backend implementation provides the operational core of the BenchPilot interface. It connects the frontend to authentication, MongoDB metadata, user workspaces, YAML generation, experiment and batch execution, monitoring, result retrieval, Docker inspection, and Jupyter notebook creation. By coordinating these components, the backend transforms the original command-line-oriented BenchPilot workflow into a managed web-based experimentation system while preserving compatibility with the existing BenchPilot execution pipeline.

### 3.5.4 Frontend Implementation

This section presents the main frontend pages of the BenchPilot interface and explains how they support experiment management from the user’s perspective. The demonstration is given from an administrator account, since the administrator can access all interface sections. Differences between administrator and regular user access are mentioned where relevant.

### 3.5.4.1 Authentication and Welcome Page

The welcome page acts as the entry point to the BenchPilot interface. As shown in Figure 13, existing users can log in using their username and password, while new users can create an account through the sign-up form shown in Figure 14. This supports the multi-user design of the system by ensuring that experiments, batches, files, and metadata are associated with the correct user account.

After successful authentication, the backend returns an HTTP-only session cookie, which is used for subsequent authenticated requests. The frontend then redirects the user to the main interface, where the experiment-management features become available.

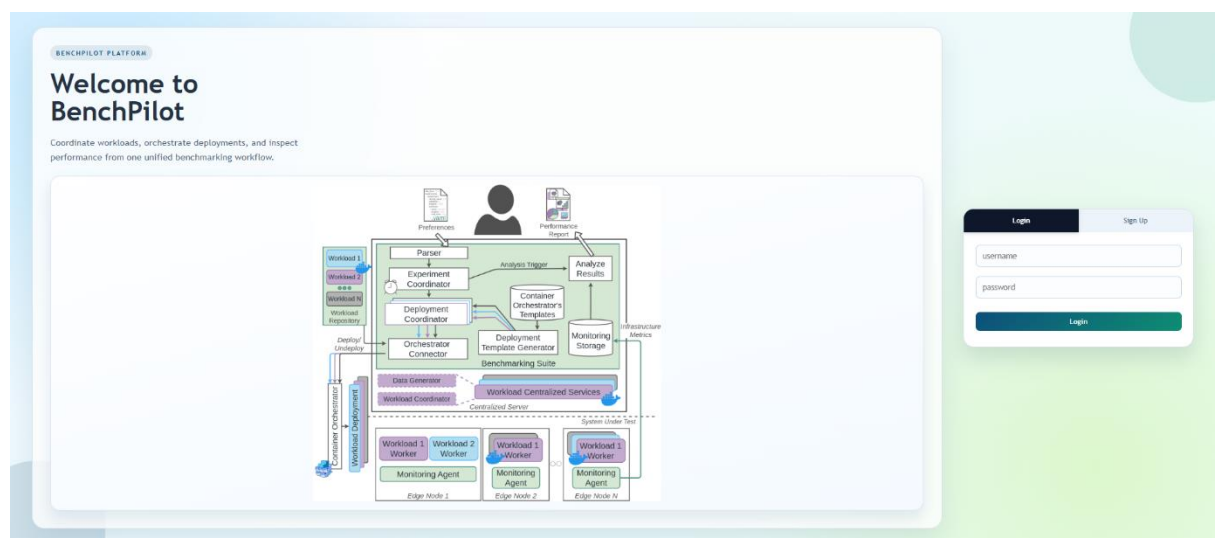


Figure 13. Login and welcome page of the BenchPilot interface

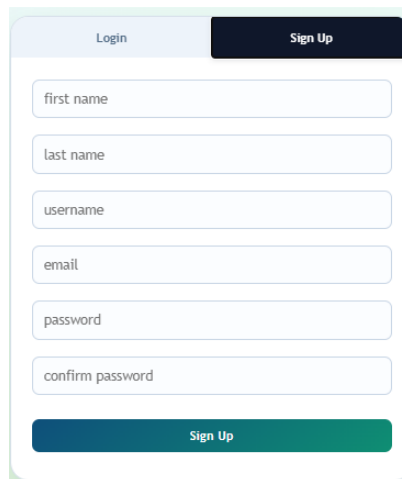
The image shows a sign-up form for the BenchPilot interface. At the top, there are two tabs: 'Login' and 'Sign Up'. The 'Sign Up' tab is active and highlighted in dark blue. Below the tabs, there are six input fields stacked vertically: 'first name', 'last name', 'username', 'email', 'password', and 'confirm password'. Each field has a light blue border and placeholder text. At the bottom of the form, there is a large, dark blue button labeled 'Sign Up' in white text.

*Figure 14. Sign-up form of the BenchPilot interface*

### **3.5.4.2 Dashboard Page**

After logging in, the user is redirected to the main dashboard, shown in Figure 15. The header displays the logged-in username on the left side and provides the main navigation options on the right side, including Deploy Experiments, View Experiments, and Logout. This allows users to move between the main experiment-management pages or end their session from any part of the interface.

The dashboard provides access to the main management sections of the system. The Clusters tab displays the cluster overview, including the manager node and the available worker nodes. Users can also navigate to the Workloads section to inspect or manage workload configuration. The Users section is available only to administrator accounts and is not shown to regular users. This separation ensures that user-management functionality remains restricted to administrators.

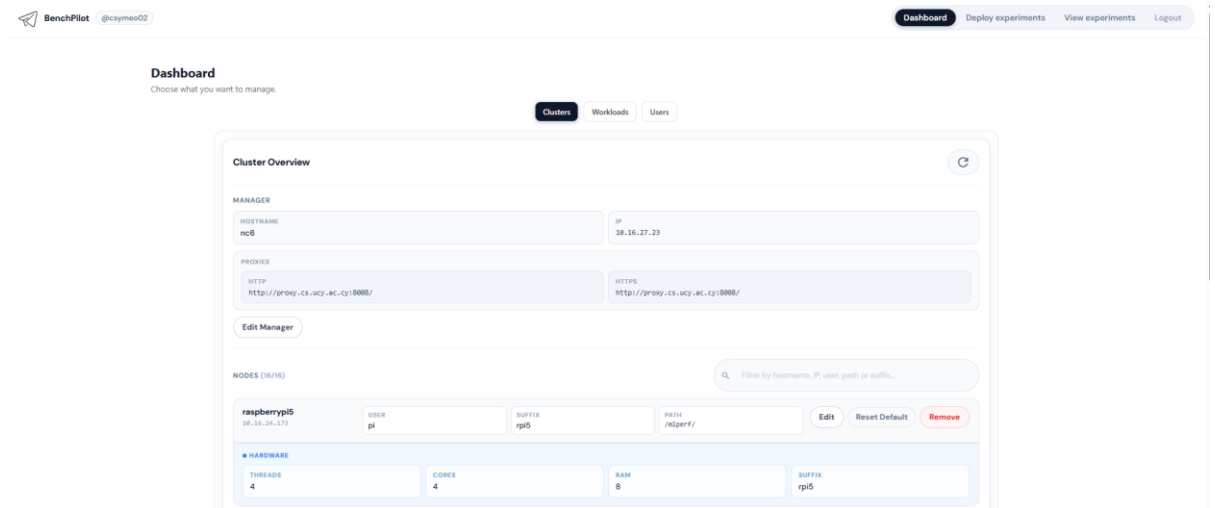


Figure 15. Dashboard page of the BenchPilot interface

## Clusters

The cluster management interface allows users to inspect and manage the edge nodes used by BenchPilot experiments. As shown in Figure 16, the cluster overview first displays the manager node, including information such as the hostname, IP address, and proxy settings. The available worker nodes are then listed separately, as shown in Figure 17. For each worker node, the interface displays the node name, IP address, username, suffix, path, and hardware-related information such as the number of threads, cores, and RAM.

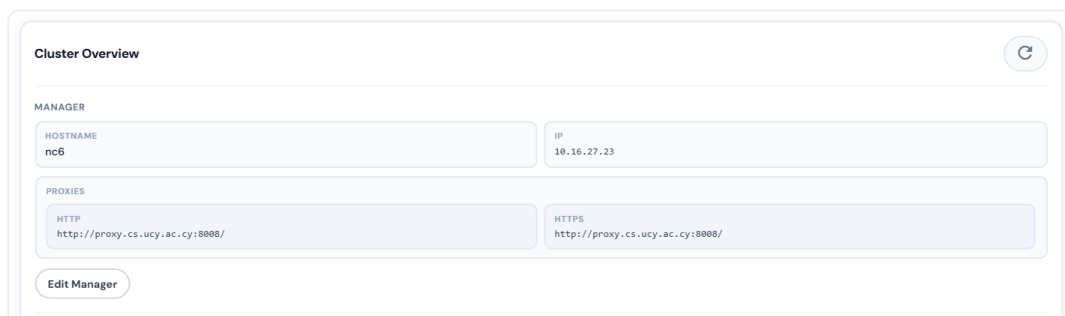
The available actions depend on the role of the logged-in user. Administrator accounts can edit the manager node information, including the hostname, IP address, proxy settings, and HTTPS configuration. Regular users do not have access to this functionality. Administrators can also add new worker nodes through the form shown in Figure 18, where the required node information can be provided, including the IP address, hostname, username, password, SSH key path, smart plug, path, threads, cores, RAM, suffix, and proxy settings.

Both administrator and regular user accounts can search the node list to locate a specific node more easily. The interface also supports resetting node information to its default values. However, editing permissions differ between roles. As shown in Figure 19,

administrators can edit all node fields and modify infrastructure-level information. In contrast, regular users can only edit the fields shown in Figure 20, which are mainly related to experiment-specific hardware and execution configuration, such as path, threads, cores, RAM, and suffix.

This separation is important because the cluster definition is based on the original BenchPilot cluster configuration file, while user-specific node information is stored separately in MongoDB when a user account is created. As a result, regular users can adjust hardware-related values for experimentation purposes without affecting the configuration used by other users. At the same time, the system can reset a user's node information by reading the original default values from the BenchPilot cluster configuration. When an administrator adds or modifies a node, the shared cluster definition is updated and the corresponding node information is propagated to the user-specific node records stored in the database.

This interface simplifies cluster management while preserving separation between shared infrastructure configuration and user-specific experiment settings.



The screenshot displays a web interface titled "Cluster Overview" with a refresh icon in the top right corner. The interface is divided into two main sections: "MANAGER" and "PROXIES".

**MANAGER**

| HOSTNAME | IP          |
|----------|-------------|
| nc6      | 10.16.27.23 |

**PROXIES**

| HTTP                            | HTTPS                           |
|---------------------------------|---------------------------------|
| http://proxy.cs.ucy.ac.cy:8080/ | http://proxy.cs.ucy.ac.cy:8080/ |

At the bottom of the Manager section, there is a button labeled "Edit Manager".

*Figure 16. Manager node information in the cluster overview*

NODES (16/16) Filter by hostname, IP, user, path or suffix...

|                                                                       |                |                  |                                |                 |               |               |        |
|-----------------------------------------------------------------------|----------------|------------------|--------------------------------|-----------------|---------------|---------------|--------|
| <b>raspberrypi5</b><br>10.16.24.173                                   | USER<br>pi     | SUFFIX<br>rpi5   | PATH<br>/mlperf/               | Edit            | Reset Default | Remove        |        |
| <b>HARDWARE</b><br>THREADS: 4    CORES: 4    RAM: 8    SUFFIX: rpi5   |                |                  |                                |                 |               |               |        |
| <b>jetson1</b><br>10.16.24.54                                         | USER<br>jetson | SUFFIX<br>jsl    | PATH<br>/mnt/sdcard/mlperf/    | Edit            | Reset Default | Remove        |        |
| <b>HARDWARE</b><br>THREADS: 4    CORES: 4    RAM: 4    SUFFIX: jsl    |                |                  |                                |                 |               |               |        |
| <b>xavier</b><br>10.16.24.182                                         | USER<br>xavier | SUFFIX<br>xavier | PATH<br>/mnt/testmount/mlperf/ | Edit            | Reset Default | Remove        |        |
| <b>HARDWARE</b><br>THREADS: 4    CORES: 6    RAM: 8    SUFFIX: xavier |                |                  |                                |                 |               |               |        |
| <b>orin</b><br>10.16.24.170                                           | USER<br>orin   | SUFFIX<br>orin   | PATH<br>/mnt/sdcard/mlperf/    | SMART PLUG<br>9 | Edit          | Reset Default | Remove |
| <b>HARDWARE</b><br>THREADS: 8    CORES: 12    RAM: 64    SUFFIX: orin |                |                  |                                |                 |               |               |        |
| <b>raspberrypi0</b><br>10.16.24.163                                   | USER<br>pi     | SUFFIX<br>rpi0   | PATH<br>/mlperf/               | Edit            | Reset Default | Remove        |        |
| <b>HARDWARE</b><br>THREADS: 4    CORES: 4    RAM: 8    SUFFIX: rpi0   |                |                  |                                |                 |               |               |        |

< 1 2 3 4 >

+ Add Node

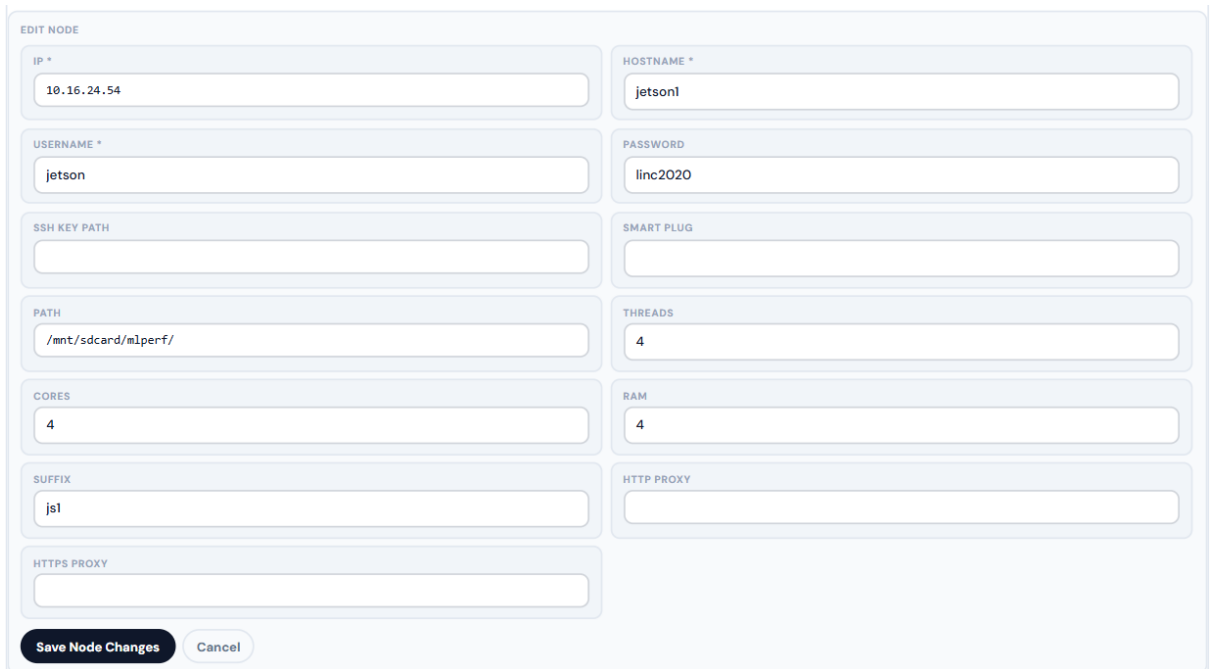
Figure 17. Worker node list in the cluster overview

ADD NODE

|                      |                      |
|----------------------|----------------------|
| IP *                 | HOSTNAME *           |
| <input type="text"/> | <input type="text"/> |
| USERNAME *           | PASSWORD             |
| <input type="text"/> | <input type="text"/> |
| SSH KEY PATH         | SMART PLUG           |
| <input type="text"/> | <input type="text"/> |
| PATH                 | THREADS              |
| <input type="text"/> | <input type="text"/> |
| CORES                | RAM                  |
| <input type="text"/> | <input type="text"/> |
| SUFFIX               | HTTP PROXY           |
| <input type="text"/> | <input type="text"/> |
| HTTPS PROXY          |                      |
| <input type="text"/> |                      |

Save Node Cancel

Figure 18. Add node form available to administrators

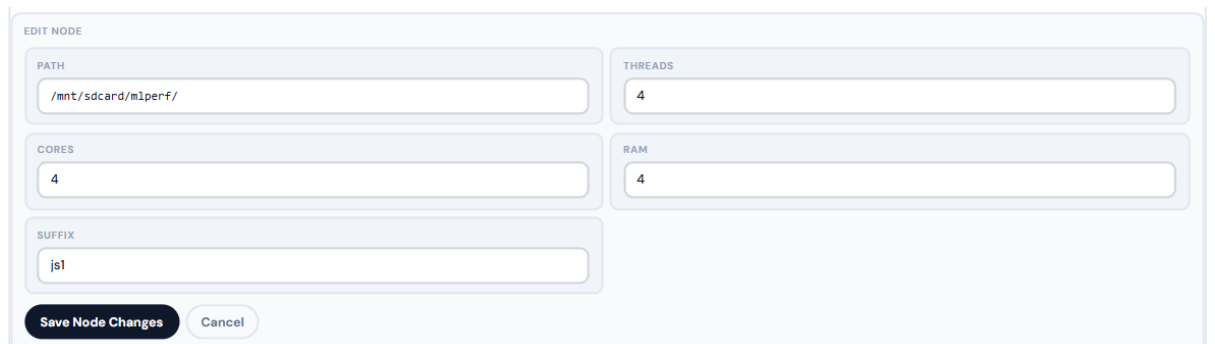


EDIT NODE

|              |                     |            |          |
|--------------|---------------------|------------|----------|
| IP *         | 10.16.24.54         | HOSTNAME * | jetsonl  |
| USERNAME *   | jetson              | PASSWORD   | linc2020 |
| SSH KEY PATH |                     | SMART PLUG |          |
| PATH         | /mnt/sdcard/mlperf/ | THREADS    | 4        |
| CORES        | 4                   | RAM        | 4        |
| SUFFIX       | jsl                 | HTTP PROXY |          |
| HTTPS PROXY  |                     |            |          |

Save Node Changes Cancel

Figure 19. Administrator node editing form



EDIT NODE

|        |                     |         |   |
|--------|---------------------|---------|---|
| PATH   | /mnt/sdcard/mlperf/ | THREADS | 4 |
| CORES  | 4                   | RAM     | 4 |
| SUFFIX | jsl                 |         |   |

Save Node Changes Cancel

Figure 20. Regular user node editing form

## Workloads

The workload management interface allows users to inspect and modify the workload templates used during experiment generation. As shown in Figure 21, the workloads are organized into separate categories, such as MLPerf, MLPerf Streaming, Database, Simple, Marketing Campaign, and Configs. Each workload template is displayed in a

structured format and includes actions for editing the template or resetting it to the default version.

When a user account is created, the default workload templates are copied into the user's MongoDB record. This means that each user has an independent set of workload templates and can modify them without affecting the templates of other users. If needed, a user can reset a modified workload back to its default configuration.

The main difference between administrator and regular user access concerns the Configs tab. Regular users can edit and reset their own workload templates, while administrators can also modify user-specific configuration values, such as server IP addresses, ports, and the default experiment structure.

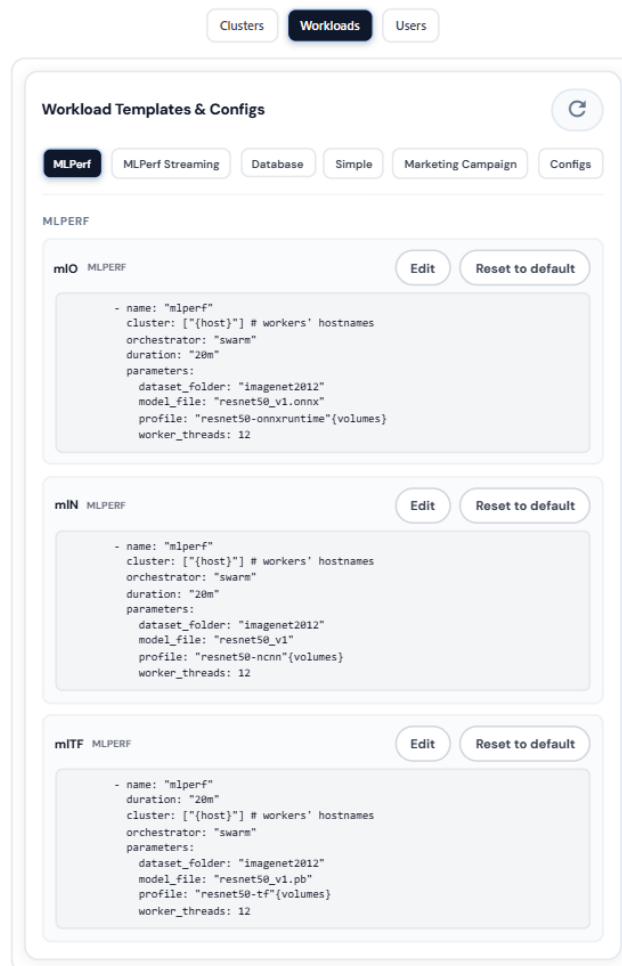


Figure 21. Workload template management interface

## Users

The user management interface is available only to administrator accounts. As shown in Figure 22, this section displays the registered users of the BenchPilot interface, including their username, email address, first name, last name, and assigned role. Administrators can also search or filter the user list to locate specific accounts more easily.

From this section, administrators can delete existing users or create new accounts. The add-user form, shown in Figure 23, allows the administrator to provide the username, password, email address, first name, last name, and role of the new user. This functionality supports controlled access to the platform and ensures that users can be managed directly through the interface without modifying the database manually.

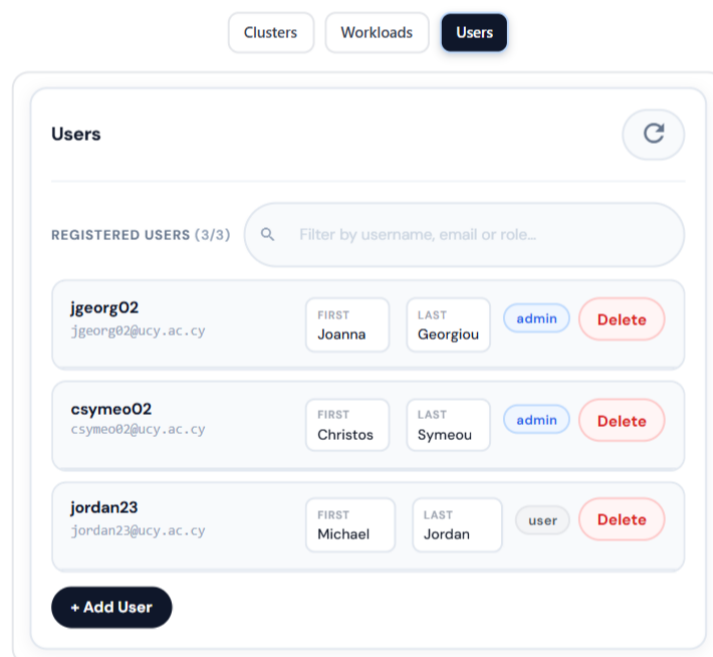


Figure 22. User management interface for administrator accounts

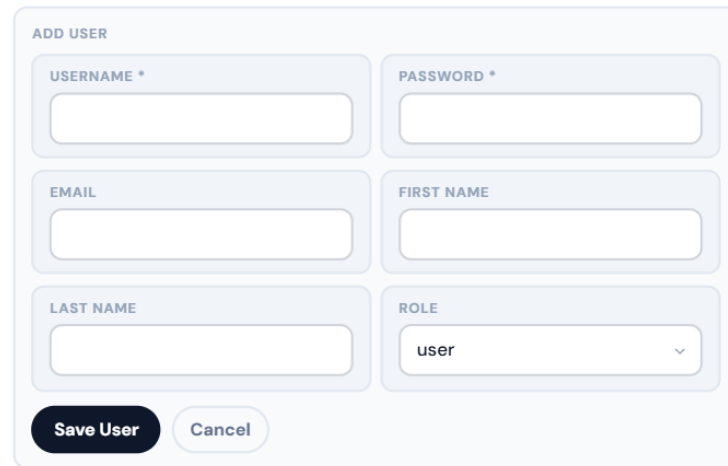
The image shows a web form titled "ADD USER" in a light blue header. The form contains six input fields arranged in a 3x2 grid. The first row has "USERNAME \*" and "PASSWORD \*", both with asterisks indicating they are required. The second row has "EMAIL" and "FIRST NAME". The third row has "LAST NAME" and "ROLE". The "ROLE" field is a dropdown menu with "user" selected and a downward arrow. At the bottom left are two buttons: "Save User" (dark blue with white text) and "Cancel" (light blue with dark blue text).

*Figure 23. Add-user form for administrator accounts*

### 3.5.4.3 Deploy Experiments Page

The deploy experiments page, shown in Figure 24, provides the main interface for creating and launching BenchPilot experiments. Its purpose is to allow users to configure experiments through the web interface instead of manually creating YAML files and executing deployment commands from the terminal.

The page is organized into three main cards. The first card is the experiment configuration card, where users define the experiment to be deployed and add it to the deployment queue. The second card is the queue card, which displays the experiments that have been selected for execution as part of a batch. This allows users to review the selected experiments before starting the deployment. The third card is the running experiments card, which shows the experiments that are currently active in the system.

This structure supports both individual and batch deployment. A user can configure and deploy a single experiment directly, or add multiple experiments to the queue and submit them as a batch. The running experiments view also provides deployment awareness by showing live experiments, helping users avoid accidentally launching conflicting workloads while another experiment is already active.

Below, a walkthrough is provided to show how a user can deploy experiments and to demonstrate the full functionality of the Deploy Experiments tab.

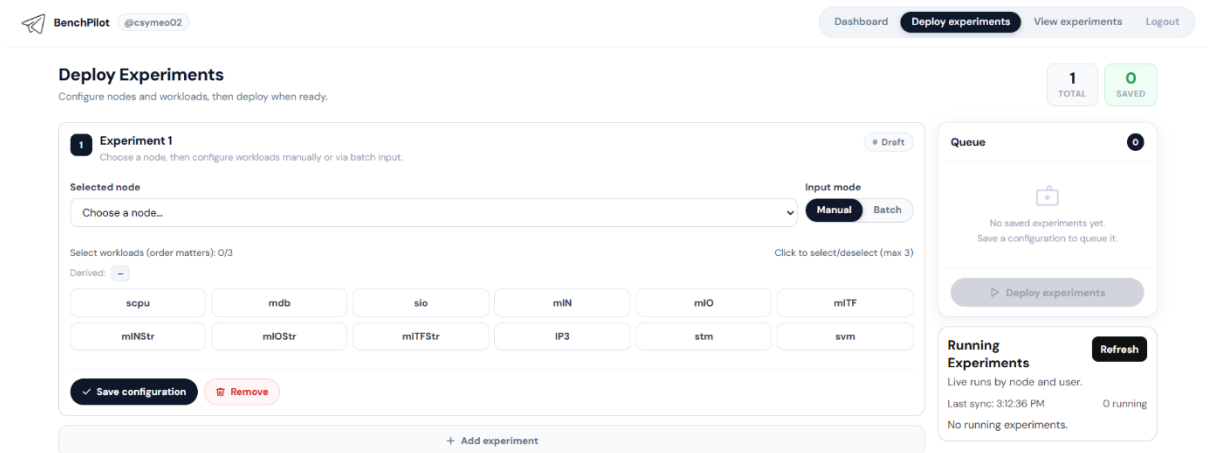


Figure 24. Deploy Experiments Page

## Experiment Deployment Walkthrough

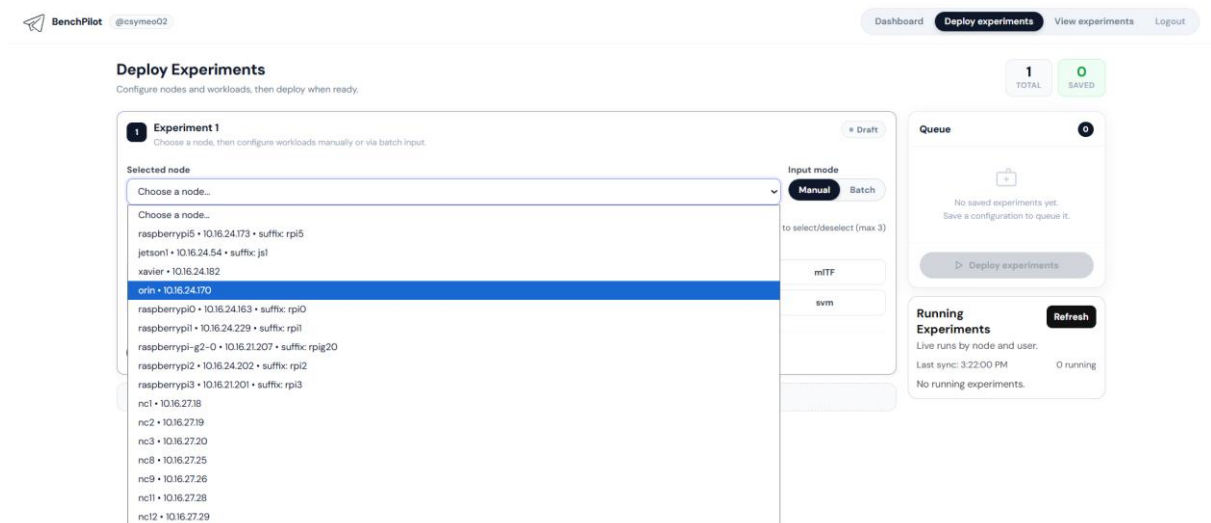


Figure 25. Node selection in the deploy experiments page

To add an experiment to the deployment queue, the user first selects the target node on which the experiment will run. As shown in Figure 25, clicking Choose a node opens a list of the available nodes configured in the system. This list corresponds to the same worker nodes presented in the dashboard cluster overview, allowing the user to select the edge device that will host the experiment.

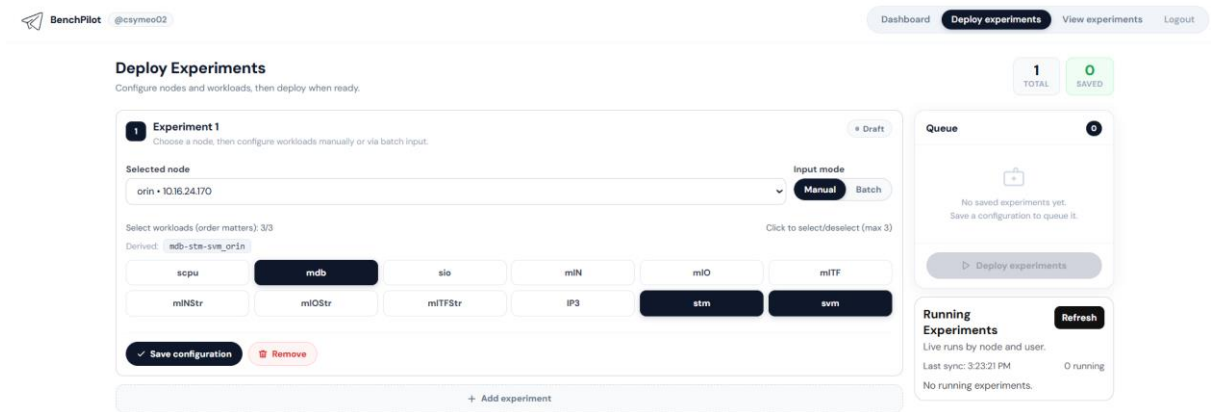


Figure 26. Manual workload selection for an experiment

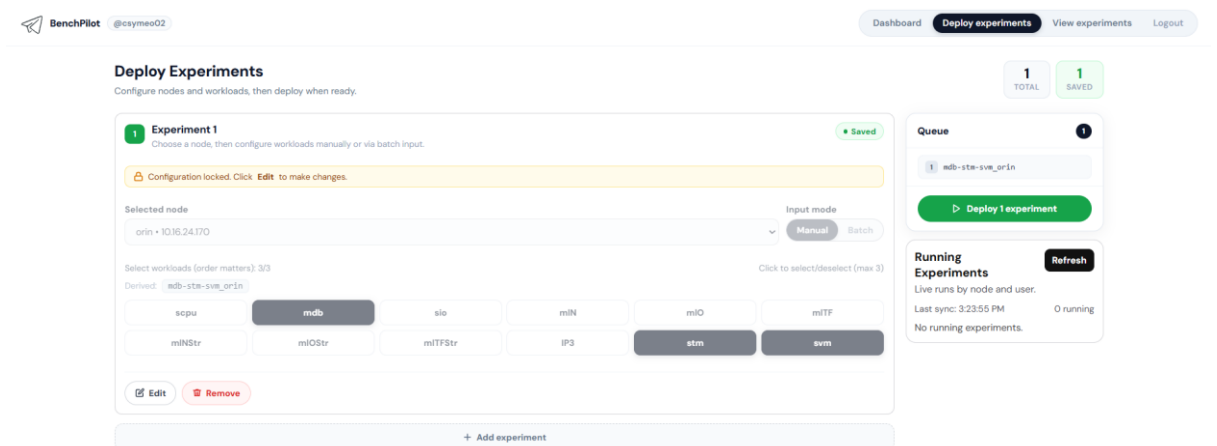


Figure 27. Saved experiment added to the deployment queue

After selecting the target node, the user can configure the experiment either manually or through batch input mode. Figure 26 shows the manual configuration mode, where the user selects the workloads that will be co-located on the chosen node. After the workload combination is selected, the user can click Save configuration to lock the configuration and add the experiment to the deployment queue. As shown in Figure 27, the saved experiment then appears in the queue, and the user can either edit the configuration, remove it, or proceed with deployment.

Select workloads (order matters): 3/3 Click to select/deselect (max 3)

Derived: `mdb-stm-svm_or1n`

scpu **mdb** sio miN miO miTF  
miNStr miOStr miTFStr IP3 **stm** svm

Edit Remove

**Experiment 2** Draft  
Choose a node, then configure workloads manually or via batch input.

Selected node: `raspberrypi5 • 10.16.24.173 • suffix: rpi5` Input mode: Manual **Batch**

**Batch experiments**  
One experiment per line (max 3 parts per line). Separators: - or \_

mdb  
mdb-scpu  
mdb-sio  
mdb-IP3  
mdb-stm

Parsed: 5

Save configuration Remove

+ Add experiment

**Queue** 1

1 `mdb-stm-svm_or1n`

Deploy 1 experiment

**Running Experiments** Refresh  
Live runs by node and user.  
Last sync: 3:28:00 PM 0 running  
No running experiments.

Figure 28. Batch input mode for adding multiple experiments

OFIN • 10.16.24.173 Experiment 2

Select workloads (order matters): 3/3 Click to select/deselect (max 3)

Derived: `mdb-stm-svm_or1n`

scpu **mdb** sio miN miO miTF  
miNStr miOStr miTFStr IP3 **stm** svm

Edit Remove

**Experiment 2** Saved  
Choose a node, then configure workloads manually or via batch input.

Configuration locked. Click Edit to make changes.

Selected node: `raspberrypi5 • 10.16.24.173 • suffix: rpi5` Input mode: Manual **Batch**

**Batch experiments**  
One experiment per line (max 3 parts per line). Separators: - or \_

mdb  
mdb-scpu  
mdb-sio  
mdb-IP3  
mdb-stm

Parsed: 5

Edit Remove

**Queue** 6

1 `mdb-stm-svm_or1n`  
2 `mdb_rpi5`  
3 `mdb-scpu_rpi5`  
4 `mdb-sio_rpi5`  
5 `mdb-IP3_rpi5`  
6 `mdb-stm_rpi5`

Deploy 6 experiments

**Running Experiments** Refresh  
Live runs by node and user.  
Last sync: 3:28:30 PM 0 running  
No running experiments.

Figure 29. Queue updated with experiments from batch input mode

The user can also add multiple experiments to the queue. As shown in Figure 28, the second experiment is configured using Batch input mode, where the user can enter several experiment names at once, with one experiment per line. In this example, five experiments are added for the selected node. After the user clicks Save configuration, the queue is updated with the five experiments declared in batch input mode, as shown in Figure 29.

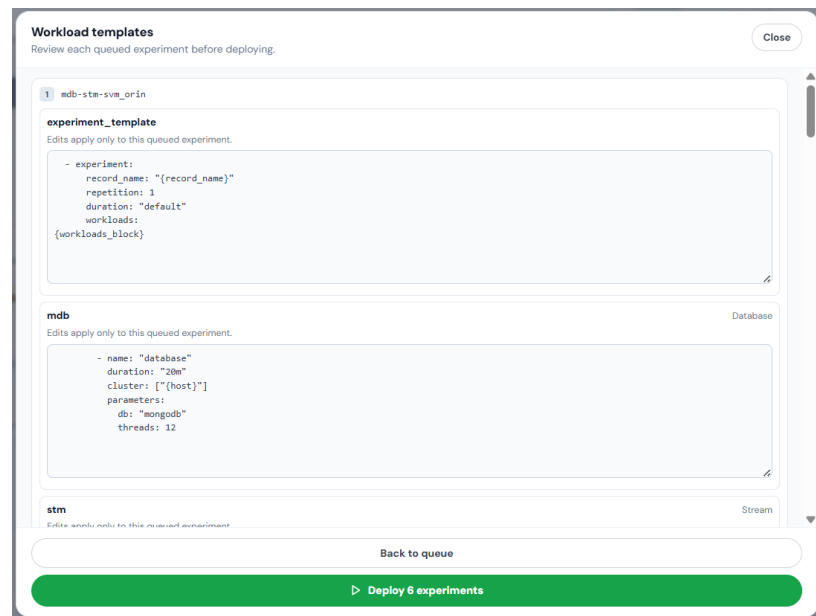


Figure 30. Workload template review before experiment deployment

Before deploying the experiments, the user can review and adjust the workload templates that will be used for the queued experiments. When the user clicks the Deploy experiments button, a pop-up window appears, as shown in Figure 30. This window displays the workload templates for each queued experiment and allows the user to make experiment-specific edits before deployment. These edits apply only to the selected queued experiment and do not modify the default workload templates stored for the user in MongoDB.

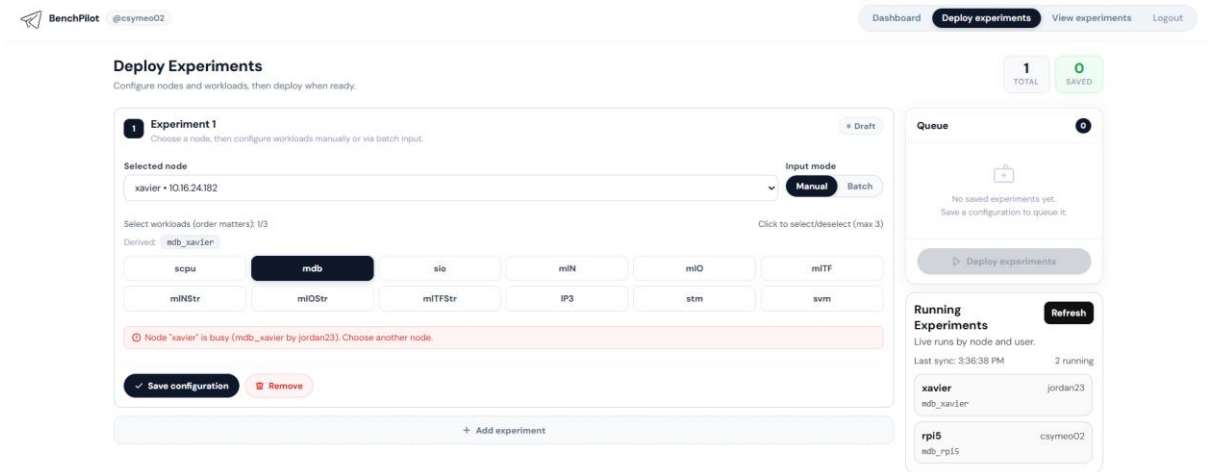


Figure 31. Deployment conflict warning for a busy node

The deploy experiments page also helps prevent conflicting deployments. As shown in Figure 31, two experiments are already running on different nodes, one on xavier and one on rpi5, by different users. When the current user selects a node that is already in use, the interface displays a warning message and prevents the experiment from being added or deployed on that node. This helps avoid accidental resource conflicts and ensures that multiple users do not run overlapping experiments on the same edge device.

### 3.5.4.4 View Experiments Page

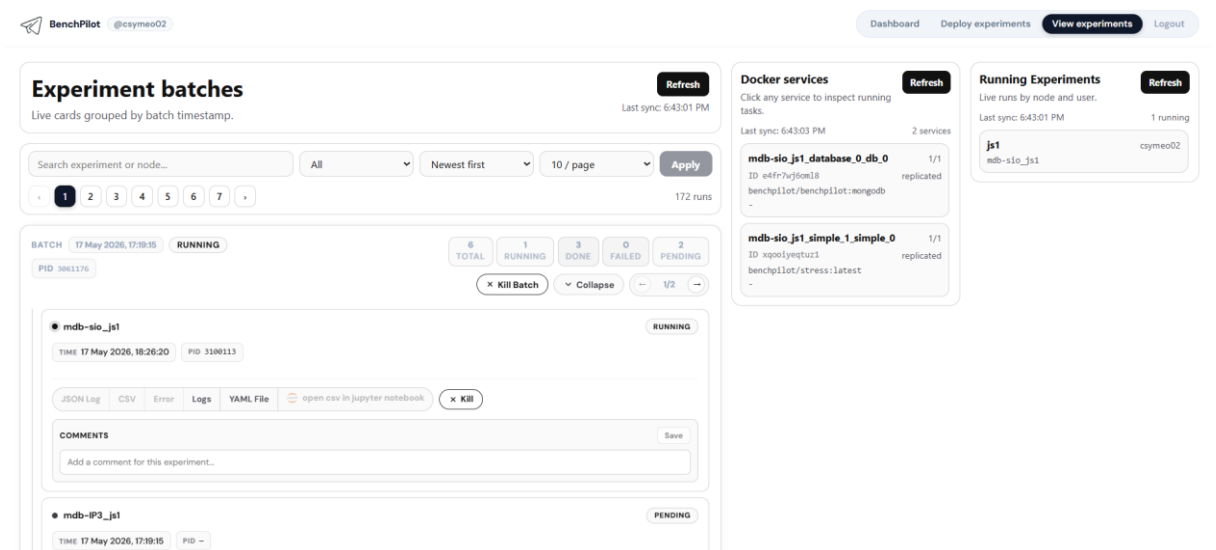


Figure 32. View Experiments Page

The View Experiments page, shown in Figure 32, allows users to monitor deployed experiments and inspect their results from a single interface. The main section displays experiment batches grouped by timestamp, together with their execution status, process identifier, and summary counters such as total, running, completed, failed, and pending experiments. Users can search, filter, sort, and paginate through previous runs, which makes the page easier to use when many experiments have been executed.

For each experiment, the interface provides access to the generated outputs, including JSON logs, CSV files, error information, execution logs, and the generated YAML file. Users can also add comments, inspect experiment status, kill a running experiment or batch, and open available CSV results directly in a generated Jupyter notebook. On the right side of the page, the interface displays the currently running Docker services and active experiments, helping users understand the live system state without manually executing Docker commands.

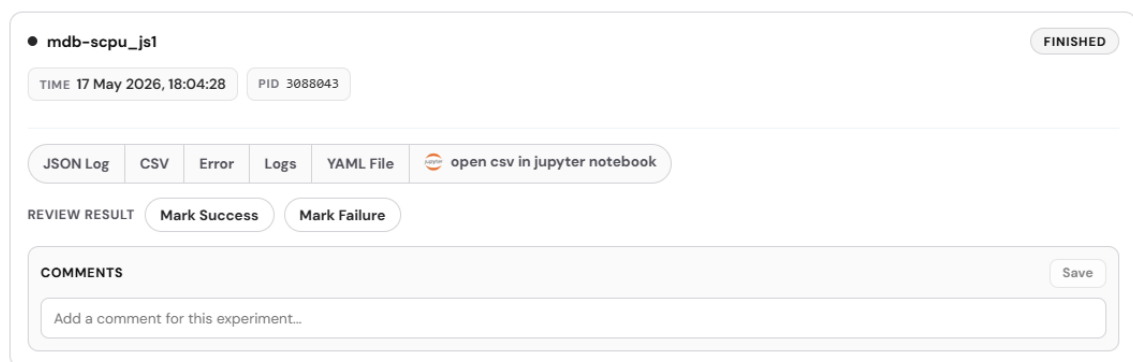


*Figure 33. Completed experiment controls and review options*

While an experiment is running, only the execution logs and the generated YAML file are available for inspection. The JSON log, CSV output, error information, and open CSV in Jupyter notebook option become available after the experiment has finished. As shown in Figure 33, completed experiments can also be reviewed manually by marking them as successful or failed. In addition, users can add comments to record observations, errors, or validation notes related to the experiment.



Figure 34. JSON log view with successful workload validation messages

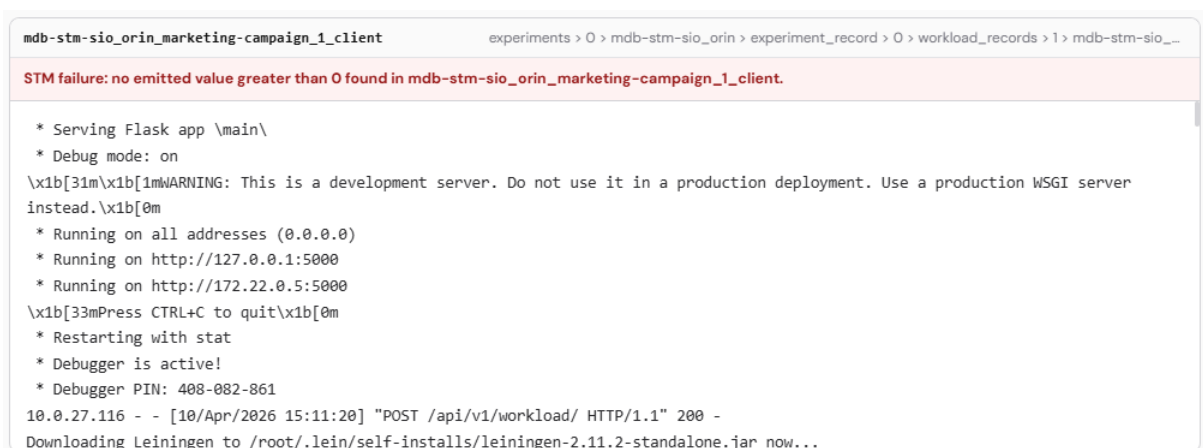


Figure 35. JSON log view with workload validation failure message

After an experiment finishes, the user can inspect the generated JSON log through the JSON Log tab. As shown in Figure 34, the interface displays the workload records together with validation messages that summarize whether the expected workload outputs were produced correctly. Successful checks are highlighted clearly, for example when the database workload produces valid YCSB (Yahoo! Cloud Serving Benchmark) output or when a stress workload starts correctly. If a workload output does not satisfy the expected condition, the interface displays a failure message, as shown in Figure 35. This helps

users identify incomplete or suspicious experiment results before using them for further analysis. The JSON log can also be downloaded directly from the interface.

CSV OUTPUT

stm\_orin.csv Download x

stm\_orin.csv 1190 rows

| temp         | orin_memory | orin_disk_writes | orin_disk_reads | orin_network_sent | orin_network_received | orin_power | orin_cpu   |
|--------------|-------------|------------------|-----------------|-------------------|-----------------------|------------|------------|
| 506666666667 | 2132.4742   | -16.0517118      | 0.0             | -259.0215538      | 5.433273              | 8.506      | 12.1698438 |
| 506666666667 | 2132.4742   | -16.0517118      | 0.0             | -259.0215538      | 5.433273              | 8.506      | 12.1698438 |
| 506666666667 | 2132.4742   | -16.0517118      | 0.0             | -259.0215538      | 5.433273              | 8.506      | 12.1698438 |
| 4            | 2130.2994   | -27.0242136      | 0.0             | -270.8549714      | 14.2175572            | 8.411      | 18.6410789 |
| 4            | 2130.2994   | -27.0242136      | 0.0             | -270.8549714      | 14.2175572            | 8.411      | 18.6410789 |
| 4            | 2130.2994   | -27.0242136      | 0.0             | -270.8549714      | 14.2175572            | 8.411      | 18.6410789 |
| 4            | 2130.2994   | -27.0242136      | 0.0             | -270.8549714      | 14.2175572            | 8.411      | 18.6410789 |
| 4            | 2130.2994   | -27.0242136      | 0.0             | -270.8549714      | 14.2175572            | 8.411      | 18.6410789 |
| 46           | 2153.753    | -1021.3018832    | 0.0             | -292.8454852      | 30.6868169            | 8.411      | 26.4913236 |
| 46           | 2153.753    | -1021.3018832    | 0.0             | -292.8454852      | 30.6868169            | 8.411      | 26.4913236 |

Showing first 300 rows out of 1190.

Figure 36. CSV output preview

CSV OUTPUT

mdb-sio\_js1.csv Download x

mdb-sio\_js1.csv 0 rows

Missing required CSV columns for resource "js1": js1\_memory, js1\_disk\_writes, js1\_disk\_reads, js1\_network\_sent, js1\_network\_received, js1\_power, js1\_cpu

CSV file is empty.

Figure 37. CSV output warning for missing required columns

The user can also inspect the CSV output produced during post-experiment analysis. As shown in Figure 36, the interface displays a preview of the CSV file, showing the first 300 rows, while also providing a download option for the complete file. The system also performs a basic validation check on the CSV structure. It verifies whether the expected columns for memory, disk writes, disk reads, network sent, network received, power, and CPU are present for the selected resource. If one or more required columns are missing,

the interface displays a warning message, as shown in Figure 37. This helps users identify incomplete or incorrectly generated CSV outputs before using them in further analysis.

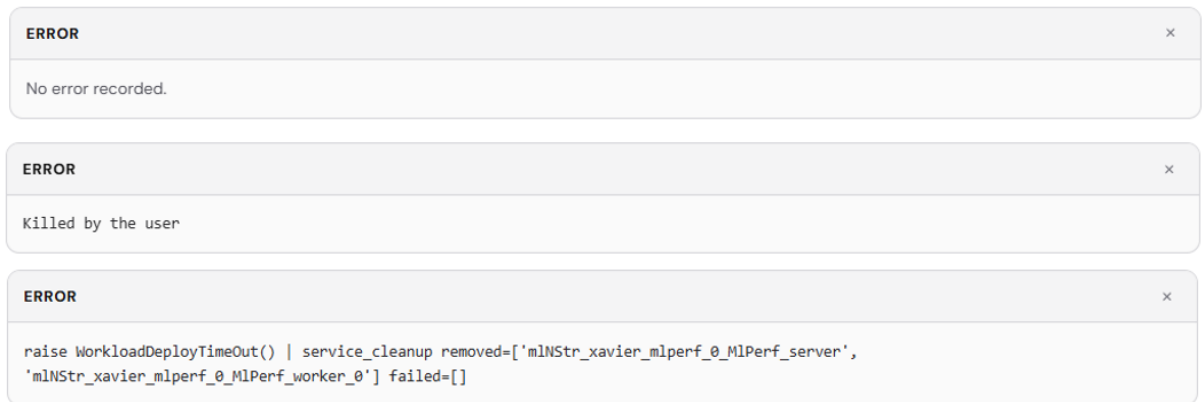


Figure 38. Error information displayed for completed or interrupted experiments

The interface also allows users to inspect error information related to each experiment. As shown in Figure 38, the Error tab displays whether an error was recorded during deployment or execution. If no error occurred, the interface reports that no error was recorded. If the experiment was stopped by the user or failed during execution, the corresponding reason is displayed, such as a user-initiated termination or a deployment timeout. This helps users understand why an experiment did not complete successfully without manually searching through execution logs.

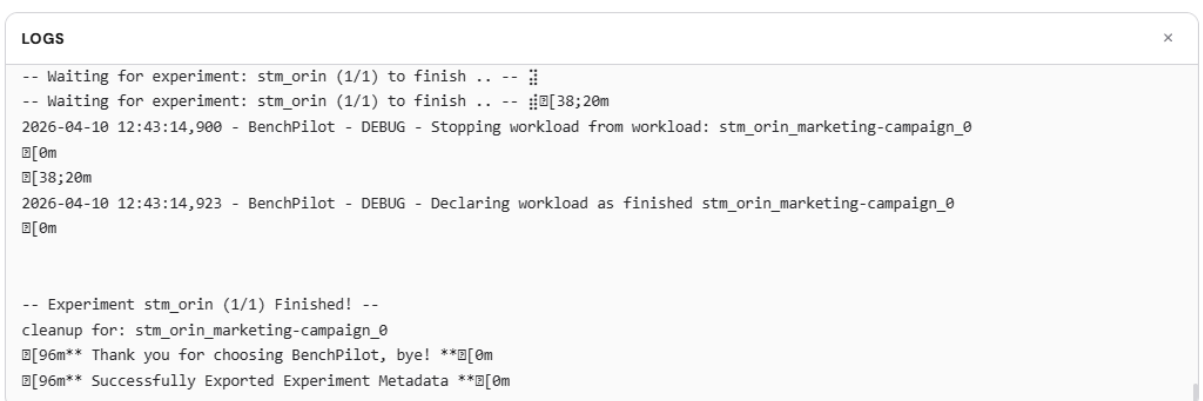
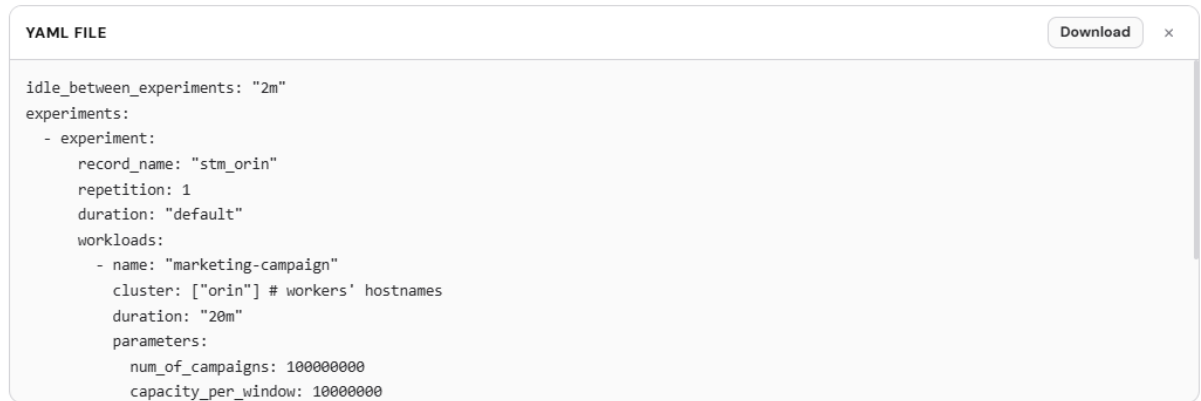


Figure 39. Execution log view for an experiment

The user can also inspect the execution logs produced while the experiment is running. As shown in Figure 39, the Logs tab displays the runtime output generated during

deployment and execution, including progress messages, workload status updates, and completion messages. This makes it easier for users to verify whether the experiment is progressing as expected and to identify possible issues without manually opening log files from the server.



```
YAML FILE Download x

idle_between_experiments: "2m"
experiments:
  - experiment:
      record_name: "stm_orin"
      repetition: 1
      duration: "default"
      workloads:
        - name: "marketing-campaign"
          cluster: ["orin"] # workers' hostnames
          duration: "20m"
          parameters:
            num_of_campaigns: 100000000
            capacity_per_window: 10000000
```

*Figure 40. Generated YAML configuration file for an experiment*

The user is also able to view and download the generated YAML file for a specific experiment. As shown in Figure 40, the YAML File tab displays the configuration that was produced by the interface and used during deployment. This allows users to inspect the exact experiment definition, including the experiment name, duration, selected workloads, and workload parameters, without manually locating the YAML file in the workspace.

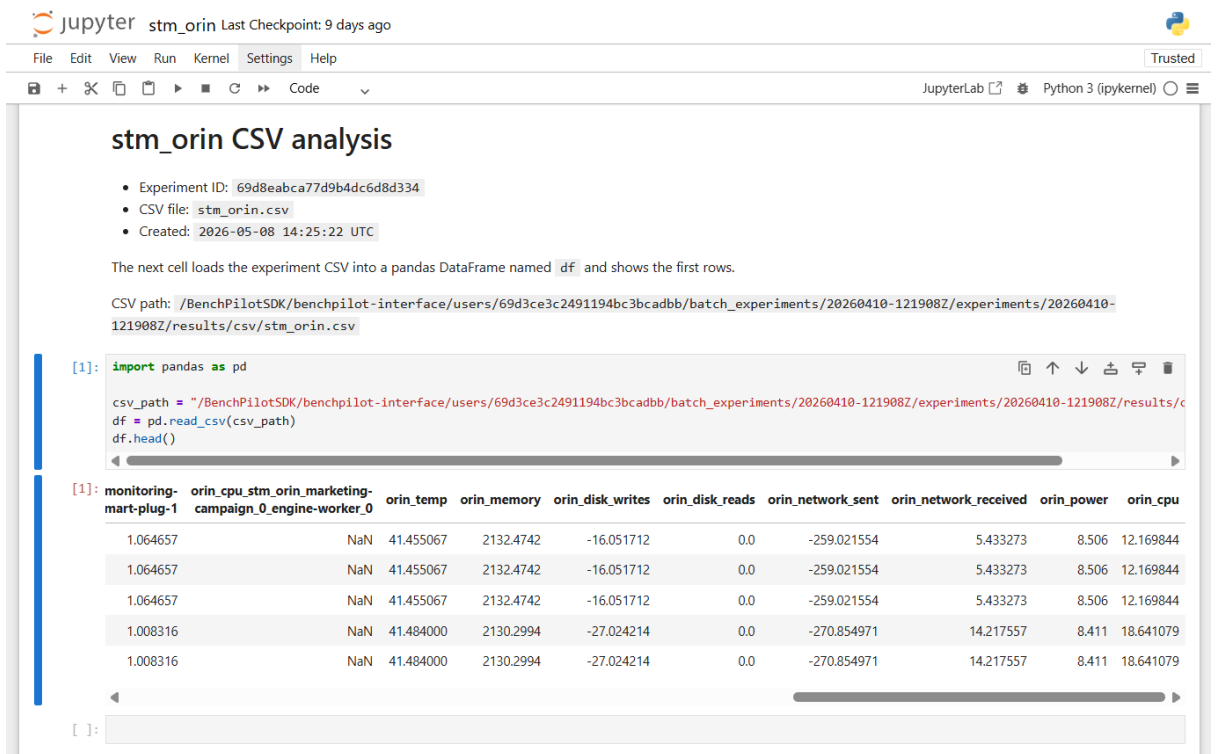


Figure 41. Generated Jupyter notebook for CSV analysis

When the user selects open CSV in Jupyter notebook, the interface opens a generated notebook for the completed experiment. As shown in Figure 41, the notebook includes basic experiment information, the path of the CSV file, and Python code that loads the CSV into a pandas DataFrame. This allows the user to continue post-experiment analysis directly in Jupyter without manually locating the CSV file or writing the initial loading code.

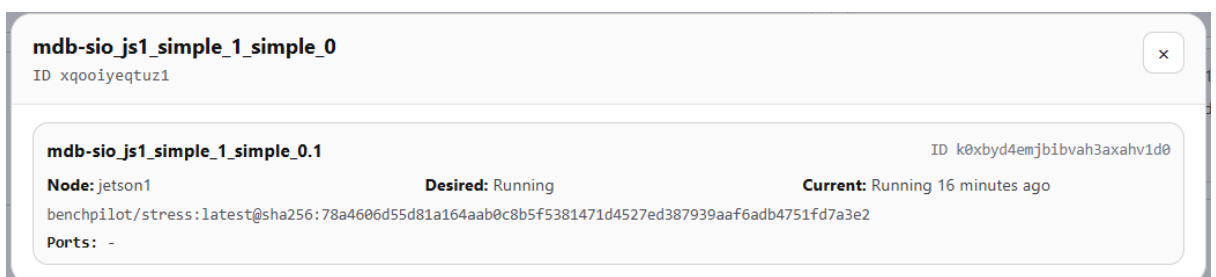


Figure 42. Docker service task details for a running experiment

Users can also inspect the Docker services associated with a running experiment. When a service is selected from the Docker services panel, the interface opens a detailed view showing the task information for that service, as shown in Figure 42. This includes the task name, task identifier, node on which the task is running, desired state, current state, image, and exposed ports when available. This view helps users understand the Docker-level execution state of an experiment without manually running Docker inspection commands in the terminal.

#### **3.5.4.5 Summary**

The frontend implementation provides the main interaction layer of the BenchPilot interface. Through the welcome page, dashboard, deployment page, workload and cluster management views, and experiment results page, users can manage the main experimentation workflow from a single web-based environment. The interface allows users to configure experiments, submit individual or batch deployments, monitor running experiments, inspect Docker services, access logs and generated files, review completed runs, and open CSV outputs in Jupyter notebooks.

By presenting these functions through structured pages and role-based access, the frontend reduces the need for manual file editing, terminal commands, and direct navigation through output directories. It also improves usability by making experiment status, configuration, results, and validation information easier to access and understand. Overall, the frontend implementation makes the BenchPilot workflow more accessible by bringing experiment configuration, deployment, monitoring, result inspection, and post-experiment analysis into a single web-based interface.

## Chapter 4: Evaluation

Chapter 3 described the implementation of the workload fingerprint extraction pipeline, the construction and preparation of the application-specific datasets, the regression-based performance modelling process, the feature importance analysis, and the clustering approach. It also presented the design and implementation of the extended BenchPilot interface for automating experiment configuration, deployment, monitoring, and result management. Building on this implementation, this chapter presents and discusses the evaluation results for both the fingerprint-based performance analysis and the developed BenchPilot interface.

The evaluation is organized into two main parts. The first part focuses on the four application-specific datasets: database performance, machine learning inference, streaming latency, and streaming tuples. For each dataset, the chapter evaluates the predictive performance of the regression models, analyses the most influential features and resource groups, and discusses the clustering results used to characterize workload behaviour. The second part evaluates the BenchPilot interface by examining how it supports the experimentation workflow, reduces manual effort, and enables more scalable and reproducible experiment management. Through this evaluation, the chapter assesses the effectiveness of workload fingerprints for performance prediction and workload characterization, while also demonstrating the practical benefits of the proposed interface in multi-tenant edge experimentation.

### 4.1 Dataset Overview

As described in Section 3.2.3, the workload fingerprint extraction pipeline produces a JSON file containing all generated fingerprints. During dataset construction, this file is transformed into four application-specific CSV datasets, corresponding to database performance, machine learning inference, streaming latency, and streaming tuples. These

datasets are used independently during model training and evaluation because each application metric represents a different type of performance target.

Figure 43 presents the complete set of columns contained in each application-specific dataset. Each dataset contains 37 columns, organized into metadata fields, the target variable, hardware features, workload identity indicators, resource utilization features, and fluctuation features. The metadata fields include information such as the application metric, workload name, filename, and resource identifier. The target variable is the Performance column, while the hardware features describe the device characteristics, including the number of cores, processor frequency, RAM, storage, and CPU capacity. The workload identity indicators represent the active workloads as binary values, and the resource utilization features summarize input/output, memory, network, CPU, and power usage. Finally, the fluctuation features describe the variability of each resource category using categorical labels.

Before training the models, a diagnostic report was generated for each dataset. This report was used to inspect the dataset shape, column data types, missing values, and basic descriptive statistics. The diagnostics confirmed that no missing values were present in any of the generated datasets. Therefore, no imputation or row removal was required before the model training stage.

Figure 44 summarizes the main characteristics of the four datasets. The database dataset contains 181 records, the machine learning inference dataset contains 352 records, and both streaming datasets contain 182 records. All datasets contain the same 37-column structure, but they differ in the scale and meaning of the performance target. For example, streaming tuples are measured at a much larger numerical scale than database performance, machine learning inference, or streaming latency. This confirms the need to treat each application metric as a separate prediction task rather than combining all records into a single model.

Overall, the dataset overview confirms that the constructed fingerprint datasets are complete, consistently structured, and suitable for the performance modelling, feature importance, and clustering analyses presented in the following sections.



Figure 43. Dataset Columns Overview

## Dataset Overview

| Dataset           | Application Metric | Records | Columns | Mean Performance | Missing Values |
|-------------------|--------------------|---------|---------|------------------|----------------|
| Database          | db.csv             | 181     | 37      | 7,915.04         | 0              |
| ML inference      | ml.csv             | 352     | 37      | 6,315.69         | 0              |
| Streaming latency | stm_latency.csv    | 182     | 37      | 1,423.99         | 0              |
| Streaming tuples  | stm_tuples.csv     | 182     | 37      | 27,207,860       | 0              |

Figure 44. Dataset Overview

## 4.2 Regression Model Performance

Section 3.2 described the data preparation process, the regression models used, and the evaluation procedure followed for each application-specific dataset. This section presents the regression results and compares how effectively the models predict workload performance for each application metric.

Since database performance, machine learning inference, streaming latency, and streaming tuples represent different performance targets, the results are analyzed separately for each dataset. The models are evaluated using  $R^2$  and MAE and are ranked according to their test  $R^2$  score.

### 4.2.1 Database Performance

Figure 45 presents the regression results for the database performance dataset. The best-performing model is **Extra Trees**, achieving a test  $R^2$  of **0.9646** and an MAE of **626.55**, which corresponds to approximately **7.12%** of the mean target value. This indicates that the model captures most of the variation in database performance while maintaining a relatively low prediction error.

The remaining tree-based models also perform strongly. Decision Tree, XGBoost, Random Forest, HistGradientBoosting, and Gradient Boosting all achieve  $R^2$  scores above **0.92**, showing that the fingerprint features provide useful information for predicting database performance. Linear Regression also performs reasonably well, with an  $R^2$  of **0.9152**, but its higher MAE suggests that a purely linear model is less accurate than the tree-based approaches. AdaBoost obtains the lowest  $R^2$  score in this dataset, with **0.8771**.

Overall, the results show that database performance can be predicted effectively from workload fingerprints. The strong performance of tree-based models suggests that database behaviour is influenced by non-linear interactions between workload identity, hardware characteristics, and resource utilization features.

RESULTS: fingerprints\_db.csv

|   | model                | test_R2  | test_MAE    | MAE_pct_mean |
|---|----------------------|----------|-------------|--------------|
| 3 | ExtraTrees           | 0.964582 | 626.548470  | 7.123797     |
| 1 | DecisionTree         | 0.940074 | 975.081684  | 11.086586    |
| 7 | XGBoost              | 0.935536 | 887.330105  | 10.088859    |
| 2 | RandomForest         | 0.933557 | 891.696865  | 10.138509    |
| 5 | HistGradientBoosting | 0.929046 | 888.031083  | 10.096829    |
| 4 | GradientBoosting     | 0.925564 | 960.559297  | 10.921468    |
| 0 | LinearRegression     | 0.915212 | 1175.377480 | 13.363930    |
| 6 | AdaBoost             | 0.877096 | 1048.926680 | 11.926197    |

Figure 45. Database Performance

#### 4.2.2 Machine Learning Inference Performance

Figure 46 presents the regression results for the machine learning inference dataset. The strongest result is again achieved by **Extra Trees**, with a test  $R^2$  of **0.9946** and an MAE of **646.55**, corresponding to approximately **10.77%** of the mean target value. This indicates that the model explains almost all the variation in inference performance.

Several other tree-based models also perform very well. Random Forest, Gradient Boosting, XGBoost, and Decision Tree all achieve  $R^2$  scores above **0.97**, confirming that tree-based models are highly effective for this prediction task. Although Decision Tree has a slightly lower  $R^2$  than some ensemble models, it achieves a lower MAE than Random Forest, Gradient Boosting, and XGBoost, showing that both metrics are useful when comparing model behavior.

HistGradientBoosting and AdaBoost also achieve high  $R^2$  scores, but their larger MAE values indicate higher absolute prediction errors. In contrast, Linear Regression performs considerably worse, with an  $R^2$  of **0.5673** and an MAE larger than the mean target value. This suggests that machine learning inference performance is not well represented by a simple linear relationship between the fingerprint features and the target value.

Overall, workload fingerprints provide a strong basis for predicting machine learning inference performance, especially when combined with tree-based ensemble methods.

RESULTS: fingerprints\_ml.csv

|   | model                | test_R2  | test_MAE    | MAE_pct_mean |
|---|----------------------|----------|-------------|--------------|
| 3 | ExtraTrees           | 0.994628 | 646.549522  | 10.774738    |
| 2 | RandomForest         | 0.974684 | 1280.096834 | 21.332794    |
| 4 | GradientBoosting     | 0.973586 | 1453.068945 | 24.215371    |
| 7 | XGBoost              | 0.973279 | 1141.803457 | 19.028137    |
| 1 | DecisionTree         | 0.973161 | 924.478873  | 15.406426    |
| 5 | HistGradientBoosting | 0.958907 | 2120.568586 | 35.339243    |
| 6 | AdaBoost             | 0.948850 | 3574.497478 | 59.568945    |
| 0 | LinearRegression     | 0.567341 | 7198.863733 | 119.968953   |

Figure 46. Machine Learning Inference Performance

### 4.2.3 Streaming Latency Performance

Figure 47 presents the regression results for the streaming latency dataset. Compared with database and machine learning inference performance, streaming latency is more difficult to predict, as shown by the lower  $R^2$  scores and higher relative MAE values.

The best-performing model is **Decision Tree**, with a test  $R^2$  of **0.9347** and an MAE of **371.73**, corresponding to approximately **25.21%** of the mean target value. **Gradient Boosting** performs almost identically, with a test  $R^2$  of **0.9346** and an MAE of **392.47**. These results show that both models capture a substantial part of the variation in streaming latency, although the prediction error is higher than in the previous datasets.

XGBoost also performs well, achieving an  $R^2$  of **0.9049**. The remaining models show lower performance, with Linear Regression obtaining the lowest  $R^2$  score, **0.7501**, and the highest MAE. Unlike the previous datasets, Extra Trees is not among the best-performing models, suggesting that streaming latency is influenced by different feature relationships and may be more sensitive to runtime variability or transient resource contention.

Overall, workload fingerprints can still support reasonable latency prediction, but the results indicate that streaming latency is a more variable and challenging performance metric.

RESULTS: fingerprints\_stm\_latency.csv

|   | model                | test_R2  | test_MAE   | MAE_pct_mean |
|---|----------------------|----------|------------|--------------|
| 1 | DecisionTree         | 0.934652 | 371.730607 | 25.206803    |
| 4 | GradientBoosting     | 0.934568 | 392.471230 | 26.613211    |
| 7 | XGBoost              | 0.904868 | 436.678597 | 29.610883    |
| 6 | AdaBoost             | 0.887200 | 593.015634 | 40.211992    |
| 2 | RandomForest         | 0.876753 | 453.334002 | 30.740274    |
| 3 | ExtraTrees           | 0.829193 | 469.020962 | 31.803997    |
| 5 | HistGradientBoosting | 0.805102 | 605.906561 | 41.086117    |
| 0 | LinearRegression     | 0.750060 | 718.570761 | 48.725801    |

*Figure 47. Streaming Latency Performance*

#### 4.2.4 Streaming Tuples Performance

Figure 48 presents the regression results for the streaming tuples dataset. This dataset is the most challenging prediction task among the four application metrics, as even the best-performing model leaves a larger portion of variance unexplained compared with the previous datasets.

The best-performing model is XGBoost, achieving a test  $R^2$  of **0.8165** and an MAE of approximately **3.40 million tuples**, which corresponds to about **14.83%** of the mean target value. Extra Trees follows closely, with a test  $R^2$  of **0.7951** and an MAE of approximately **3.23 million tuples**, corresponding to **14.07%** of the mean. These results suggest that both models provide useful predictions, although XGBoost achieves the stronger explanatory performance while Extra Trees produces a slightly lower absolute error.

The remaining models perform noticeably worse. Random Forest, AdaBoost, and HistGradientBoosting achieve  $R^2$  scores between approximately **0.66** and **0.67**, while Gradient Boosting and Linear Regression fall lower, with  $R^2$  scores of **0.6015** and **0.5779** respectively. Decision Tree performs the weakest, with an  $R^2$  of **0.3680**, indicating that a single tree is not sufficient for modelling this more complex target.

Overall, streaming tuples performance is harder to predict than the other application metrics. However, the results still show that ensemble methods, particularly XGBoost and Extra Trees, can capture useful relationships between workload fingerprints and throughput behavior.

RESULTS: fingerprints\_stm\_tuples.csv

|   | model                | test_R2  | test_MAE     | MAE_pct_mean |
|---|----------------------|----------|--------------|--------------|
| 7 | XGBoost              | 0.816539 | 3.400342e+06 | 14.826242    |
| 3 | ExtraTrees           | 0.795087 | 3.225869e+06 | 14.065501    |
| 2 | RandomForest         | 0.672928 | 4.367334e+06 | 19.042540    |
| 6 | AdaBoost             | 0.668749 | 4.993808e+06 | 21.774104    |
| 5 | HistGradientBoosting | 0.657858 | 5.087888e+06 | 22.184315    |
| 4 | GradientBoosting     | 0.601539 | 4.987685e+06 | 21.747406    |
| 0 | LinearRegression     | 0.577936 | 5.905843e+06 | 25.750777    |
| 1 | DecisionTree         | 0.367982 | 4.767886e+06 | 20.789035    |

Figure 48. Streaming Tuples Performance

#### 4.2.5 Summary of Regression Results

Figure 49 summarizes the best-performing model for each application metric. Extra Trees achieves the best results for database performance and machine learning inference, Decision Tree performs best for streaming latency, and XGBoost performs best for streaming tuples.

Across all datasets, tree-based models consistently outperform Linear Regression, showing that workload performance depends on non-linear relationships between workload identity, hardware characteristics, and resource utilization features. The results

also show that prediction difficulty varies by application metric. Database performance and machine learning inference are predicted with high accuracy, streaming latency is moderately more difficult, and streaming tuples is the most challenging target. Overall, the regression results confirm that workload fingerprints can effectively support performance prediction, especially when used with tree-based and ensemble-based regression models.

**Best Regression Model per Application Metric**  
Summary of the best-performing regression model for each application-specific dataset.

| Dataset                  | Best Model    | Test R <sup>2</sup> | MAE    | MAE % of Mean |
|--------------------------|---------------|---------------------|--------|---------------|
| Database Performance     | Extra Trees   | 0.9646              | 626.55 | 7.12%         |
| ML Inference Performance | Extra Trees   | 0.9946              | 646.55 | 10.77%        |
| Streaming Latency        | Decision Tree | 0.9347              | 371.73 | 25.21%        |
| Streaming Tuples         | XGBoost       | 0.8234              | ~3.24M | 14.13%        |

Figure 49. Summary of Best Regression Models per Application Metric

### 4.3 Feature Importance Results

After evaluating the predictive performance of the regression models, feature importance analysis is used to understand which fingerprint features contributed most to the predictions. For each application-specific dataset, the analysis is applied to the best-performing model based on the test R<sup>2</sup> score. Individual feature importance is examined using permutation importance and SHAP analysis, while grouped feature importance is used to summarize the impact of broader categories such as CPU, memory, input/output, network, power, and workload identity. This section presents the resulting importance patterns for each application metric.

### 4.3.1 Individual Feature Importance

Individual feature importance is used to identify which specific fingerprint features have the strongest influence on the prediction of each application metric. In this analysis, the best-performing regression model for each dataset is selected based on the test  $R^2$  score, and feature importance is then computed on that model.

The importance of each feature is measured using two complementary methods: permutation importance and SHAP analysis. Permutation importance measures how much the model performance decreases when the values of a single feature are randomly shuffled on the test set. A larger decrease in  $R^2$  indicates that the model depends more strongly on that feature. While permutation importance provides a clear ranking of feature influence, it does not reveal the direction of each feature's effect. To address this, SHAP analysis is also applied.

SHAP values are based on a simple idea. For a single prediction, the SHAP value of a feature describes how much that feature pushed the prediction up or down compared to the average prediction. A positive SHAP value indicates that the feature increased the predicted value, a negative SHAP value indicates that it decreased the predicted value, and a SHAP value close to zero indicates that the feature had little effect on the prediction. By aggregating these per-prediction contributions across the test set, SHAP provides a global view of feature importance that also captures the direction of each feature's effect.

In this thesis, SHAP results are presented using two complementary plots. The first is a bar plot of the mean absolute SHAP value per feature,  $\text{mean}(|\text{SHAP}|)$ , which summarises the average magnitude of each feature's effect on the model output and provides a clear ranking of the most important features. The second is the summary (beeswarm) plot, where each test sample is shown as a single dot per feature. The horizontal position of a dot represents the SHAP value of that feature for that sample, while the colour represents the feature value, ranging from blue for low values to red for high values. While the bar plot identifies which features are most influential overall, the beeswarm plot makes it

possible to observe both the magnitude and the direction of each feature's effect on the predicted performance.

The following subsections present the individual feature importance results separately for the database, machine learning inference, streaming latency, and streaming tuples application metrics.

#### **4.3.1.1 Individual Feature Importance for Database Application Metric**

##### **Permutation Importance**

Figure 50 presents the permutation importance results for the database application metric, computed on the Extra Trees model identified in Section 4.2.1 (test  $R^2 = 0.9646$ ). The values on the horizontal axis represent the average decrease in  $R^2$  when the values of a feature are randomly shuffled on the test set, so a larger decrease indicates that the model relies more strongly on that feature for prediction [30].

IO\_mean is the most influential feature by a large margin, with a permutation importance of approximately 0.253, around four times larger than that of the second most important feature. This is consistent with the nature of the database workload, which is based on the Yahoo! Cloud Serving Benchmark (YCSB) and continuously performs read, update, and insert operations on a NoSQL system.

The next most influential features are CPU-related. CPU\_mean and CPU\_peak obtain similar importance values of approximately 0.064 and 0.063, while cpu\_capacity and cores contribute approximately 0.039 and 0.022 respectively. Together, these features show that the available compute capacity of the device is the second main factor affecting database performance.

The remaining features have smaller but interpretable effects. ram and IO\_peak show moderate importance values of approximately 0.015 and 0.014, while Memory\_mean and IO\_fluctuation remain close to 0.001, indicating that average and peak resource usage are more informative than resource variability. Workload identity indicators such as sio,

which represents the disk input/output stress workload, and scpu, which represents the CPU stress workload, obtain small but non-negligible values of approximately 0.0054 and 0.0045. This shows that co-located stress workloads influence database performance, although much of their effect is captured indirectly through the resource utilization features. Other indicators, including svm, mlTF, and stm, have only marginal impact.

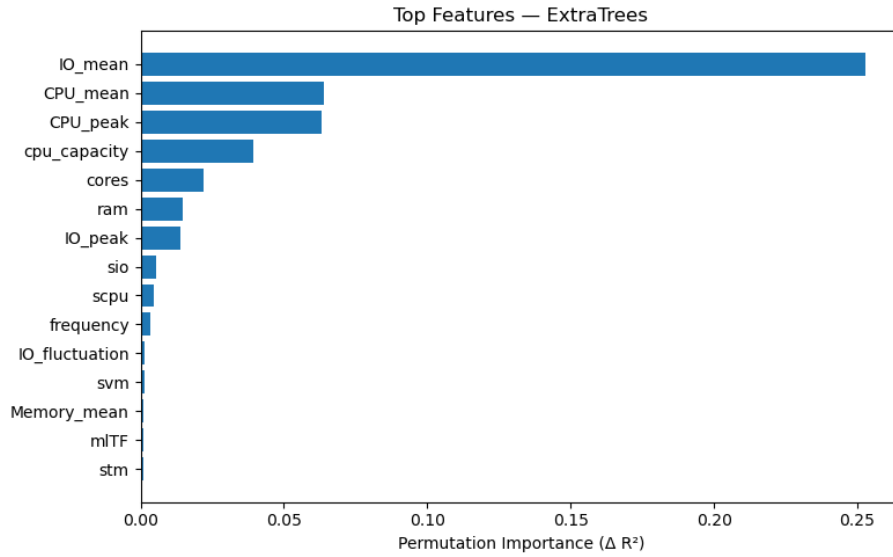


Figure 50. Permutation Importance for Database Metric

## SHAP Analysis

Figure 51 presents the SHAP summary plot for the database application metric, computed using the Extra Trees model. The ranking is consistent with the permutation importance results, with IO\_mean appearing as the most influential feature.

The most influential feature is again IO\_mean, with SHAP values ranging from approximately  $-3000$  to  $+2500$ . High values of IO\_mean clearly push the predicted database performance upward, while low values pull it downward. This confirms the dominant role of average input/output activity already observed in the permutation importance results and is consistent with the YCSB-based database workload, where throughput depends heavily on the rate of disk operations performed during execution.

The CPU utilization features show the opposite pattern. For both CPU\_peak and CPU\_mean, high feature values are associated with negative SHAP contributions, while low values are associated with positive contributions. This indicates that elevated CPU utilization reduces the predicted database performance, which is consistent with resource contention from co-located workloads. When other processes consume a large share of the CPU, the database workload completes fewer operations within the same time window.

Hardware features such as cpu\_capacity, cores, ram, and frequency display the opposite monotonic relationship. Higher feature values increase the predicted performance, while lower values decrease it. This indicates that more capable devices generally achieve higher database throughput, although the final prediction is still influenced by workload composition and resource utilization.

Among the remaining features, IO\_peak contributes moderately and follows the same direction as IO\_mean, complementing it by capturing short-lived bursts of disk activity. IO\_fluctuation remains tightly clustered around zero, confirming that input/output variability is far less informative than its average and peak levels. Workload identity indicators such as svm, sio, and scpu, which represent memory stress, disk input/output stress, and CPU stress workloads respectively, have small negative contributions when the corresponding stress workload is present. This suggests that the model can detect co-location interference, although most of this effect is already captured indirectly through the resource utilization features. Finally, Power, Memory, Network, and the remaining workload indicators show SHAP values tightly distributed around zero, confirming that they have a marginal effect on database performance prediction.

Figure 52 shows the mean absolute SHAP values for the same model. The ranking is consistent with Figure 51 and the permutation importance results in Figure 50, with IO\_mean clearly dominating, followed by CPU-related and hardware-related features.

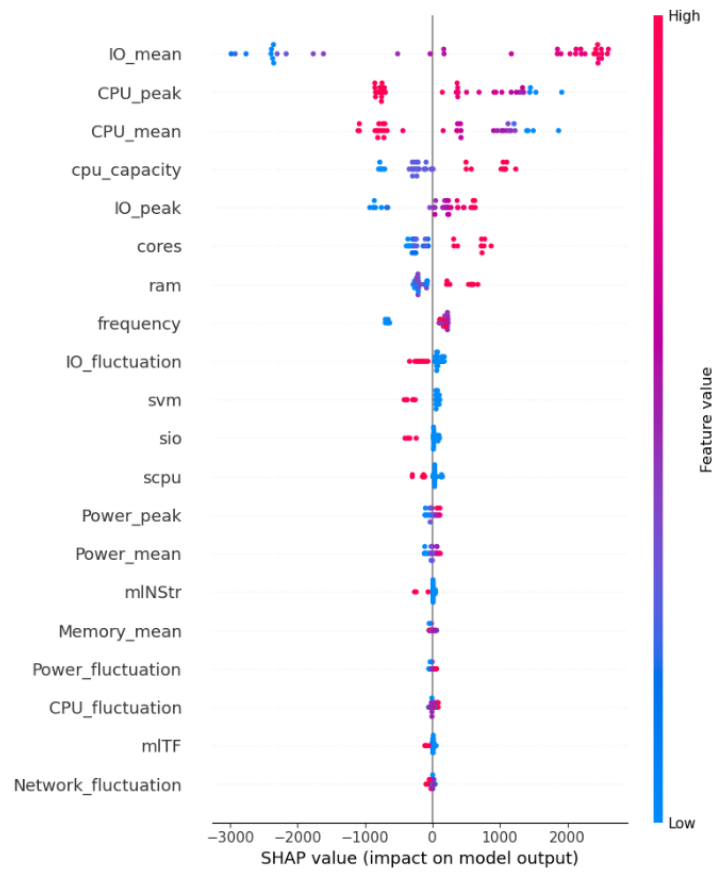


Figure 51. SHAP Summary (beeswarm) Plot for Database Metric

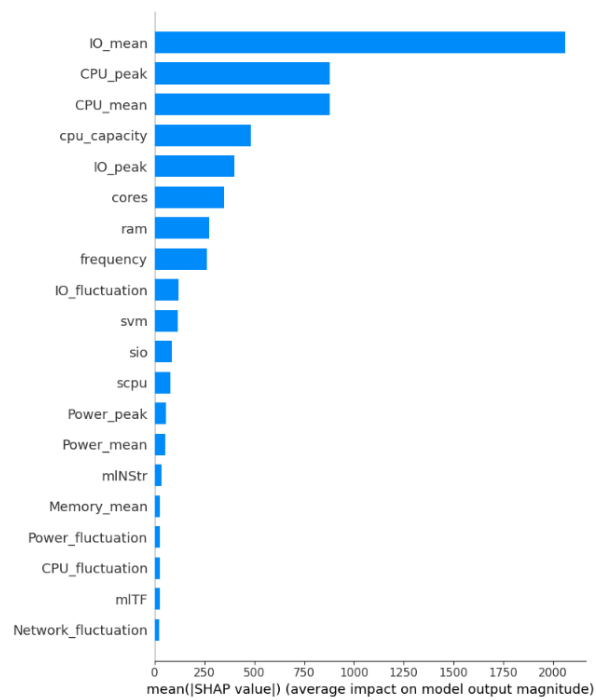


Figure 52. Mean Absolute SHAP Values for Database Metric

## Conclusion

The database application metric shows a clear dependence on input/output behaviour, especially average input/output activity. CPU utilization and hardware capacity provide additional predictive information, while workload identity features contribute less directly. However, their small negative effects indicate that co-located stress workloads still influence database throughput. These results confirm that the fingerprint representation captures both resource usage and co-location effects relevant to database performance prediction.

### 4.3.1.2 Individual Feature Importance for Machine Learning Application Metric

#### Permutation Importance

Figure 53 presents the permutation importance results for the machine learning application metric, computed using the Extra Trees model identified in Section 4.2.2 with a test  $R^2$  of 0.9946.

The most influential feature is mlO, which represents the MLPerf ONNX inference workload, with an importance value of approximately 0.395. This shows that workload identity plays a central role in predicting machine learning inference performance. The importance of mlTF, which represents the MLPerf TensorFlow inference workload and appears fifth with a value of approximately 0.084, further supports this observation, indicating that different machine learning workloads produce distinct performance patterns.

Power-related features are also highly influential. Power\_mean is the second most important feature, with an importance value of approximately 0.323, while Power\_peak reaches approximately 0.090. This suggests that power consumption provides useful information about the intensity of the inference workload during execution.

CPU-related features follow next. CPU\_mean has an importance value of approximately 0.126, and CPU\_peak contributes approximately 0.069. These results show that processor utilization is another important factor for predicting inference performance.

Compared with workload identity, power, and CPU utilization, the remaining features have much smaller effects. Hardware features such as cpu\_capacity, cores, and ram contribute only modestly, while memory-related features, CPU fluctuation, and most co-located workload indicators remain close to zero.

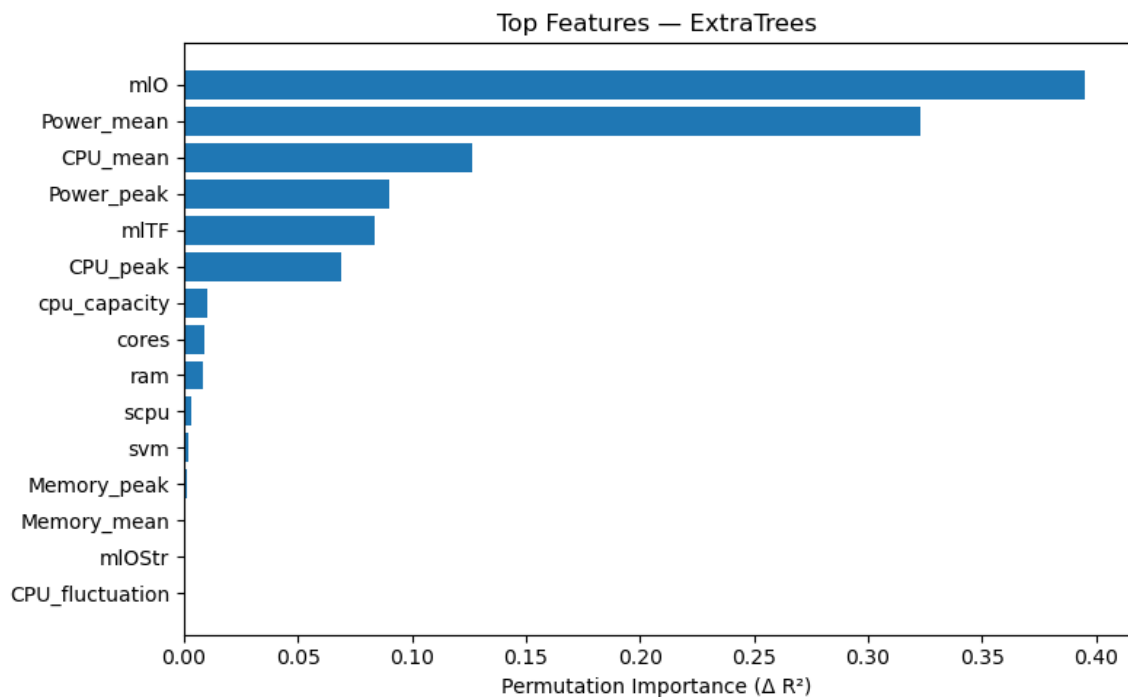


Figure 53. Permutation Importance for Machine Learning Metric

## SHAP Analysis

Figure 54 presents the SHAP summary plot for the machine learning application metric, computed using the Extra Trees model. The ranking is consistent with the permutation importance results, with mlO, which represents the MLPerf ONNX inference workload, appearing as the most influential feature. Its SHAP values range from approximately  $-5000$  to more than  $+25000$ , showing that this workload identity feature has a strong effect on the predicted inference performance. When mlO is present, it generally pushes

the prediction upward, indicating that this workload is associated with higher predicted performance values.

Power-related features are also highly influential. `Power_mean` has a wide SHAP range, extending from negative values to more than +25000, while `Power_peak` also shows clear positive contributions for higher values. This confirms that power consumption is strongly connected to the model's prediction of machine learning inference performance.

CPU-related features provide another important contribution. `CPU_mean` and `CPU_peak` appear among the top-ranked features, although their SHAP patterns are more mixed. High CPU usage does not always push the prediction in the same direction, suggesting that CPU utilization interacts with workload type and other resource measurements rather than acting as a simple independent predictor.

The feature `mlTF`, which represents the MLPerf TensorFlow inference workload, is also important, confirming that workload identity is a major factor for this application metric. Hardware-related features such as `cpu_capacity`, `cores`, and `ram` appear lower in the ranking and have smaller SHAP magnitudes, indicating that they contribute to the prediction but are less influential than workload identity, power, and CPU utilization.

The remaining features, including memory usage, fluctuation values, network behaviour, and most co-located workload indicators, are concentrated close to zero. This indicates that they have only a limited effect on the prediction of machine learning inference performance.

Figure 55 shows the mean absolute SHAP values for the same model. The ranking confirms the pattern observed in Figure 54 and in the permutation importance results, with `mlO`, `Power_mean`, `CPU_mean`, `mlTF`, `Power_peak`, and `CPU_peak` appearing as the most influential features.

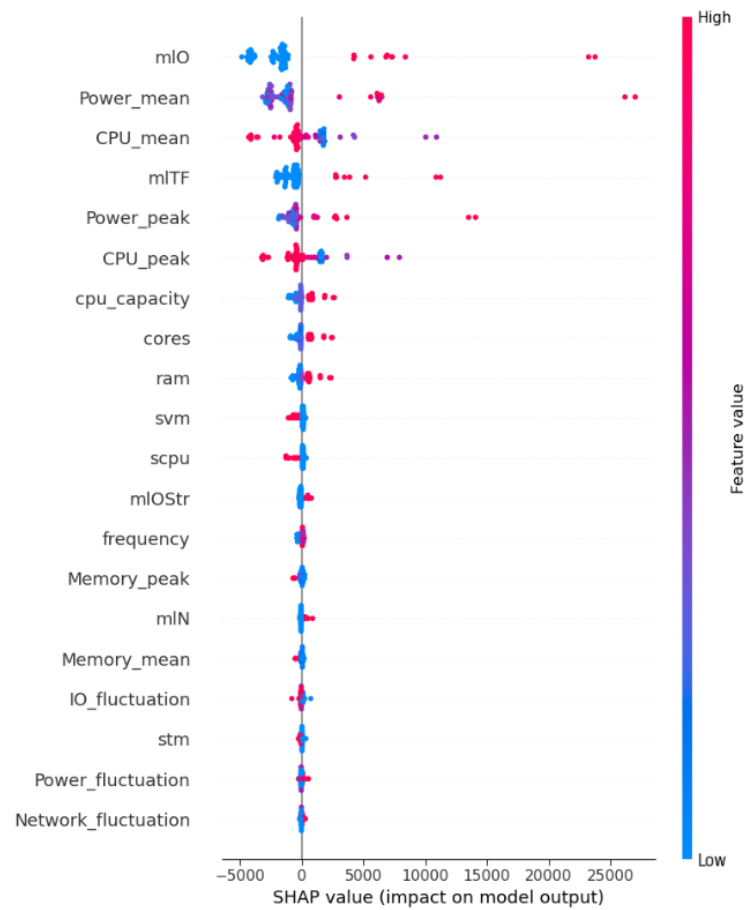


Figure 54. SHAP Summary (beeswarm) Plot for Machine Learning Metric

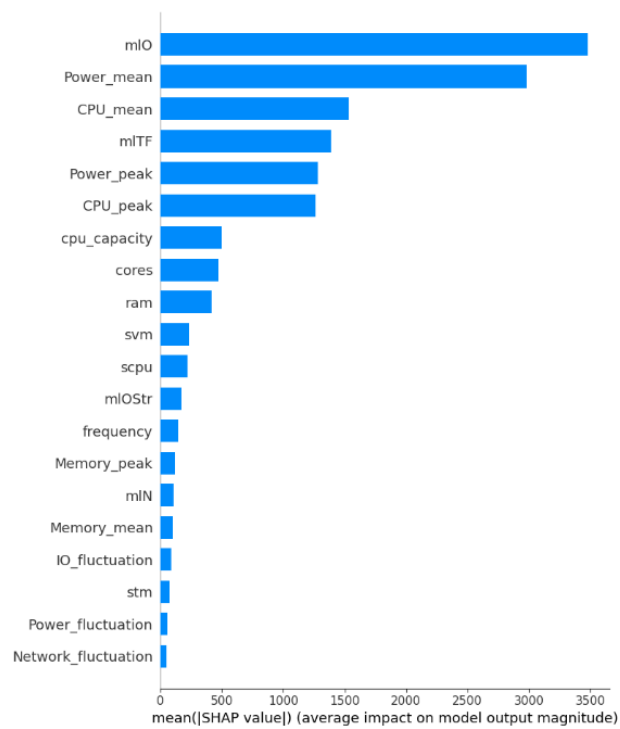


Figure 55. Mean Absolute SHAP Values for Machine Learning Metric

## Conclusion

For the machine learning application metric, the results show that workload identity is one of the main factors affecting prediction, especially through the MLPerf ONNX and TensorFlow inference workloads represented by mlO and mlTF. This indicates that different inference workloads produce distinct performance patterns that are useful for the model. Power consumption and CPU utilization also play an important role, capturing the computational intensity of the inference execution. Hardware features such as `cpu_capacity`, `cores`, and `ram` provide smaller contributions, while memory, network, fluctuation features, and most co-located workload indicators have limited influence. These results confirm that the fingerprint representation captures the main factors affecting machine learning inference performance, particularly workload type and resource intensity.

### 4.3.1.3 Individual Feature Importance for Streaming Latency Application Metric

#### Permutation Importance

Figure 56 presents the permutation importance results for the streaming latency application metric, computed using the Decision Tree model identified in Section 4.2.3 with a test  $R^2$  of 0.9347.

The results show that `Network_mean` is the dominant feature by a large margin, with an importance value of approximately 1.286. This indicates that average network activity is the most important factor for predicting streaming latency. This is expected, since streaming applications depend heavily on continuous data movement, and changes in network behaviour can directly affect the time required to process and deliver tuples through the pipeline.

The next most influential features are `Power_mean`, `cores`, and `Network_peak`, with importance values of approximately 0.207, 0.176, and 0.129, respectively. The importance of `Network_peak` reinforces the role of network activity, while `cores` shows

that the hardware capacity of the device also affects latency. Power\_mean suggests that the overall resource intensity of the execution is also informative for the model.

The workload identity feature mlTFStr, which represents the MLPerf TensorFlow workload running in streaming mode, also appears among the top features, with an importance value of approximately 0.104. This indicates that the presence of a co-located streaming inference workload has a visible effect on latency prediction. Memory\_mean follows with an importance value of approximately 0.063, showing that memory usage also contributes, but less strongly than network-related features.

The remaining features have considerably smaller importance values. CPU, input/output, fluctuation features, and most other resource indicators are close to zero, suggesting that they have limited direct influence on the prediction compared with network behaviour, power usage, and hardware capacity.

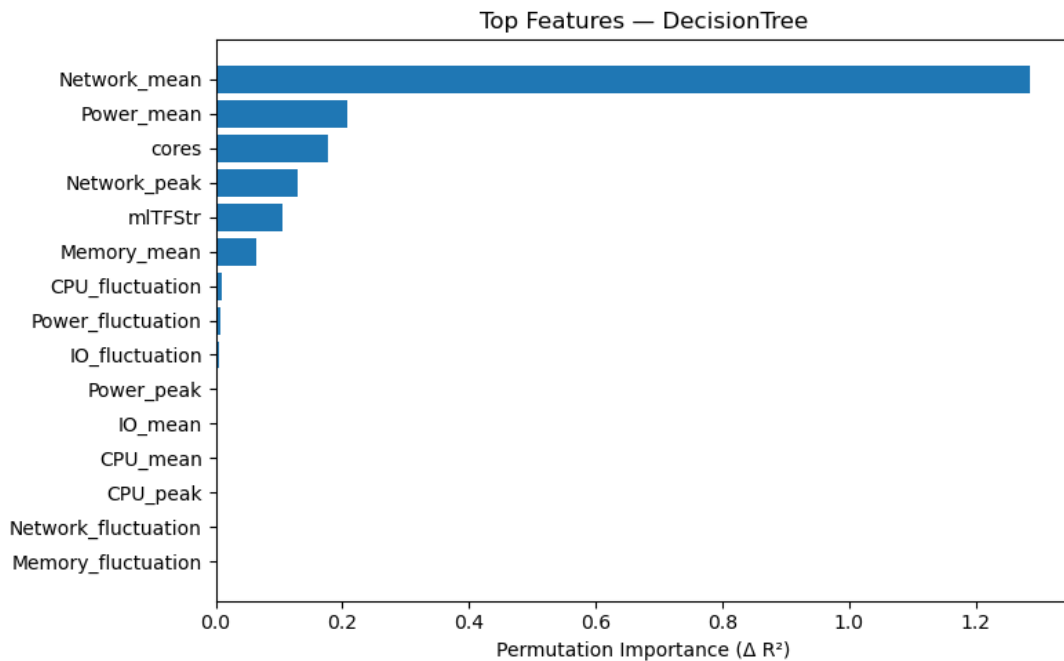


Figure 56. Permutation Importance for Streaming Latency Metric

## SHAP Analysis

Figure 57 presents the SHAP summary plot for the streaming latency application metric, computed using the Decision Tree model. The ranking is consistent with the permutation importance results, with `Network_mean` appearing as the most influential feature. Its SHAP values range from approximately  $-1000$  to more than  $+6000$ , showing that average network activity has a strong effect on the predicted latency.

The colour distribution shows that high values of `Network_mean` push the predicted latency upward, while low values tend to reduce it. This confirms that increased average network activity is associated with higher streaming latency. This is consistent with the nature of the streaming workload, where data must continuously move through the processing pipeline and network pressure can increase end-to-end delay.

The hardware feature `cores` is the second most influential feature in the SHAP ranking. Lower values of `cores` are generally associated with positive SHAP contributions, while higher values tend to reduce the predicted latency. This indicates that devices with fewer CPU cores are associated with higher predicted latency, while devices with greater compute capacity help reduce it.

`Network_peak` also appears among the most important features and follows a similar direction to `Network_mean`. Higher peak network activity tends to increase the predicted latency, showing that both sustained and short-lived network pressure affect streaming performance. `Power_mean` also contributes to the prediction, although its effect is smaller and less direct than the network-related features.

The workload identity feature `mlTFStr` appears among the top features, but with lower SHAP magnitude than the network and hardware features. This suggests that the presence of the MLPerf TensorFlow streaming workload contributes to latency variation, although much of its effect is reflected through the measured resource utilization features. Other features, including CPU utilization, input/output activity, fluctuation features, and most

memory and power features, are concentrated close to zero, indicating limited influence on the prediction.

Figure 58 shows the mean absolute SHAP values for the streaming latency model. The ranking confirms the pattern observed in Figure 57 and in the permutation importance results, with Network\_mean appearing as the dominant feature, followed by cores, Network\_peak, Power\_mean, and mlTFStr.

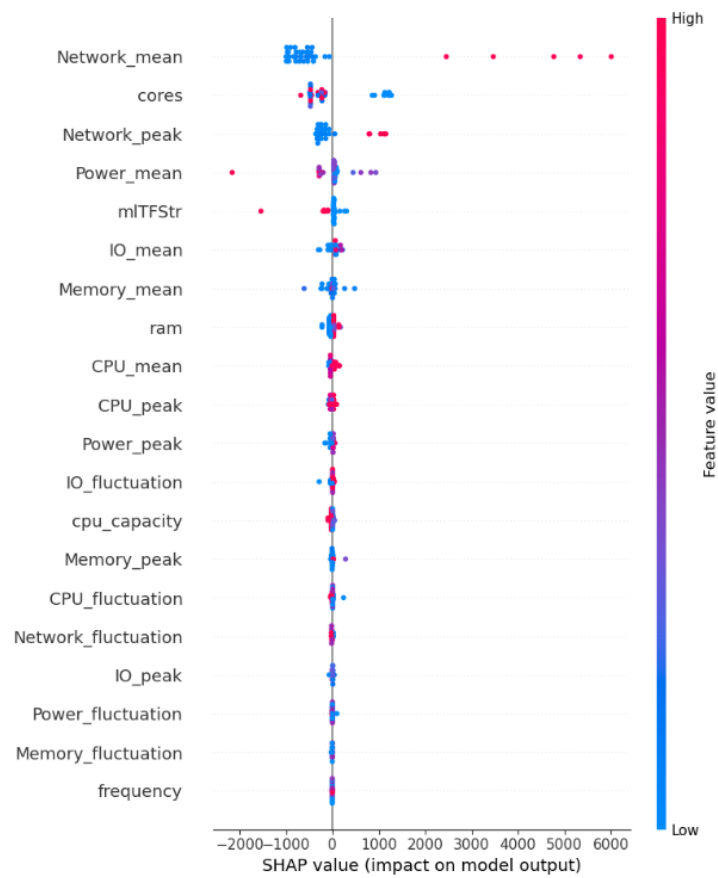


Figure 57. SHAP Summary (beeswarm) Plot for Streaming Latency Metric

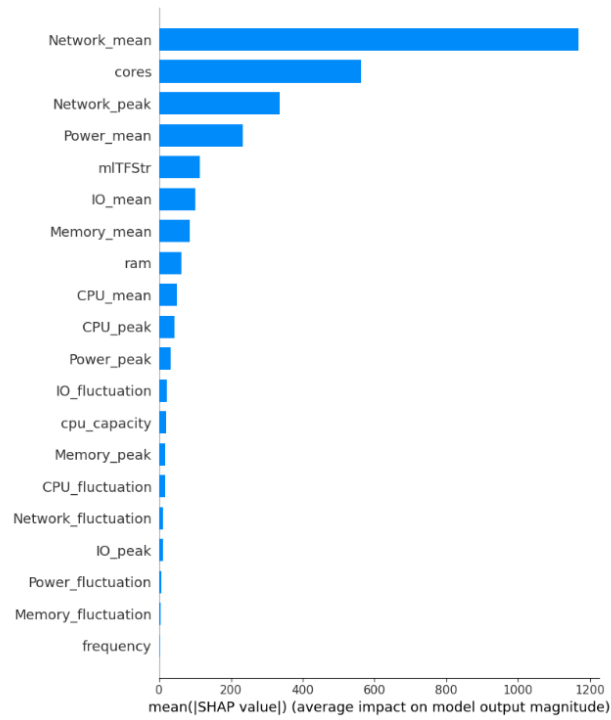


Figure 58. Mean Absolute SHAP Values for Streaming Latency Metric

## Conclusion

For the streaming latency application metric, both permutation importance and SHAP analysis highlight the importance of network behaviour, especially average network activity. Peak network usage also contributes to the prediction, while device capacity, represented by the number of cores, helps explain latency differences across resources. Power usage provides additional information about execution intensity, and the presence of mlTFStr shows that co-located streaming inference can influence latency. These results confirm that the fingerprint representation captures the main factors affecting streaming latency, particularly network pressure, hardware capacity, and selected co-location effects.

#### 4.3.1.4 Individual Feature Importance for Streaming Tuples Application Metric

##### Permutation Importance

Figure 59 presents the permutation importance results for the streaming tuples application metric, computed using the XGBoost model identified in Section 4.2.4.

The most influential feature is `Network_peak`, with an importance value of approximately 0.340, making it the dominant predictor of streaming tuples performance. This indicates that peak network activity has the strongest effect on the number of tuples processed. Since the streaming workload continuously exchanges and processes data across the pipeline, short periods of high network demand can strongly affect throughput. The second most important feature is `cores`, with an importance of approximately 0.229, showing that the available parallel processing capacity of the device also plays a major role in determining streaming throughput. `Network_mean` follows with an importance of approximately 0.113, confirming that both sustained and peak network activity are important for this application metric.

The next group of influential features includes `Power_peak` and `Power_mean`, with importance values of approximately 0.081 and 0.064, respectively. These results suggest that power consumption captures useful information about the overall execution intensity of the streaming workload. Another important feature is `IP3`, with an importance of approximately 0.062. Since `IP3` represents the `IPerf3` network stress workload, its presence indicates that co-located network-intensive traffic has a visible impact on streaming tuples performance. This is consistent with the communication-heavy nature of the streaming application. The workload identity feature `mlO`, which represents the `MLPerf ONNX` inference workload, also contributes, although more modestly, with an importance of approximately 0.030.

Several additional features have smaller but still interpretable contributions. `Network_fluctuation` and `IO_peak` both contribute around 0.025, while `IO_mean`, `Memory_peak`, and `mlOStr` remain lower, with values between approximately 0.010 and

0.015. These features suggest that, beyond network behaviour and hardware capacity, some contribution also comes from input/output activity, memory usage, and specific co-located workloads. In contrast, Memory\_mean, frequency, and sio have only minor influence, indicating that they are much less important than the dominant network-related features.

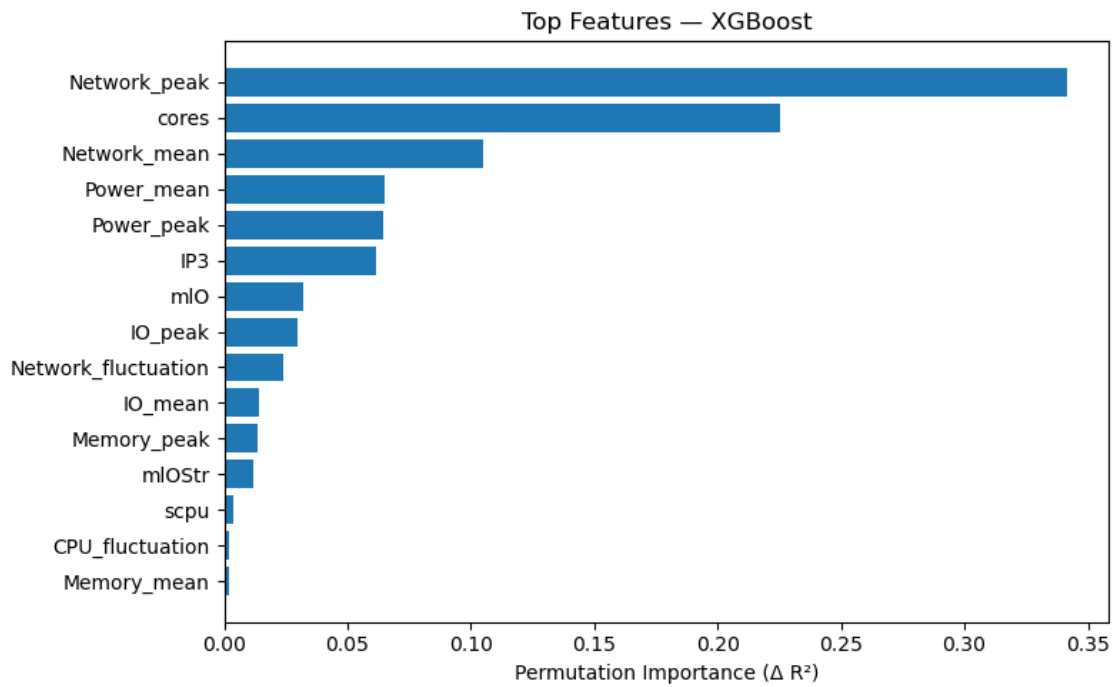


Figure 59. Permutation Importance for Streaming Tuples Metric

## SHAP Analysis

Figure 60 presents the SHAP summary plot for the streaming tuples application metric, computed using the XGBoost model. The ranking is broadly consistent with the permutation importance results, with cores, Network\_peak, Power\_mean, Power\_peak, and Network\_mean appearing as the most influential features.

The most influential feature in the SHAP ranking is cores. Higher values of cores are associated with positive SHAP contributions, while lower values tend to reduce the predicted number of processed tuples. This indicates that devices with greater parallel processing capacity are expected to achieve higher streaming throughput.

Network-related features also play a major role. `Network_peak` has large SHAP values and shows that higher peak network activity is generally associated with lower predicted tuple throughput. This suggests that short periods of high network pressure can negatively affect the streaming pipeline. `Network_mean` also contributes to the prediction, confirming that both peak and average network behaviour are important for explaining streaming tuples performance.

Power-related features, especially `Power_mean` and `Power_peak`, also appear among the most important predictors. Higher power values generally push the prediction upward, suggesting that increased power consumption reflects more intensive execution and higher processing activity during streaming.

The workload identity feature `IP3`, which represents the IPerf3 network stress workload, also contributes to the prediction. Its presence is mostly associated with negative SHAP values, indicating that co-located network stress reduces the predicted number of processed tuples. This is consistent with the communication-heavy nature of the streaming tuples metric.

The remaining features, including CPU usage, input/output usage, memory features, fluctuation values, and most other workload identity indicators, have smaller SHAP magnitudes and are concentrated closer to zero. This indicates that they have a more limited influence compared with network behaviour, device capacity, and power consumption.

Figure 61 shows the mean absolute SHAP values for the same model. The ranking confirms that `cores` is the strongest feature, followed by `Network_peak`, `Power_mean`, `Power_peak`, and `Network_mean`.

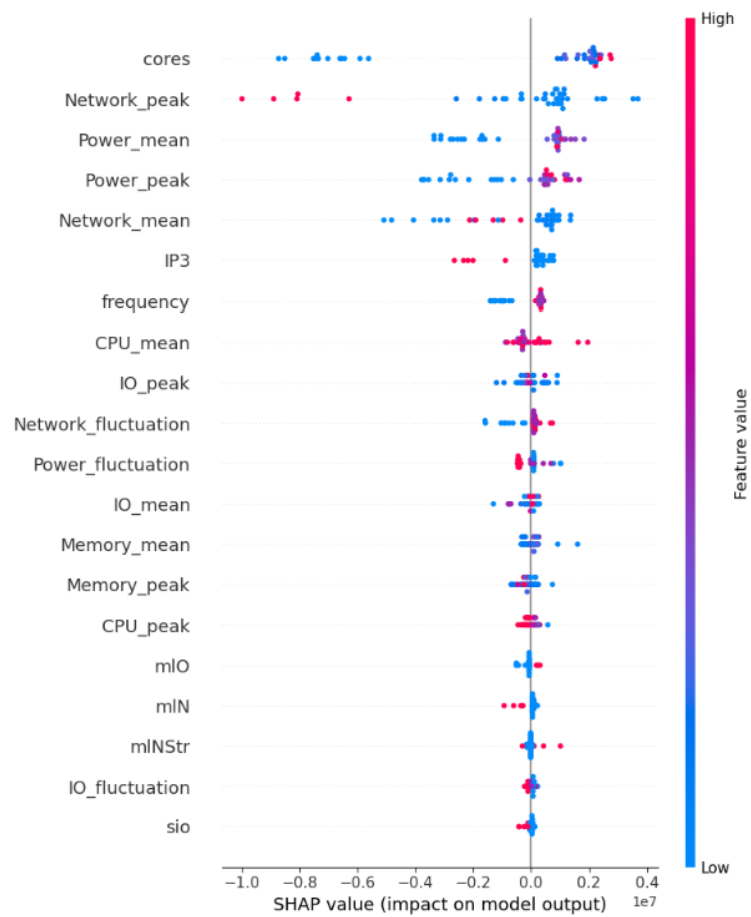


Figure 60. SHAP Summary (beeswarm) Plot for Streaming Tuples Metric

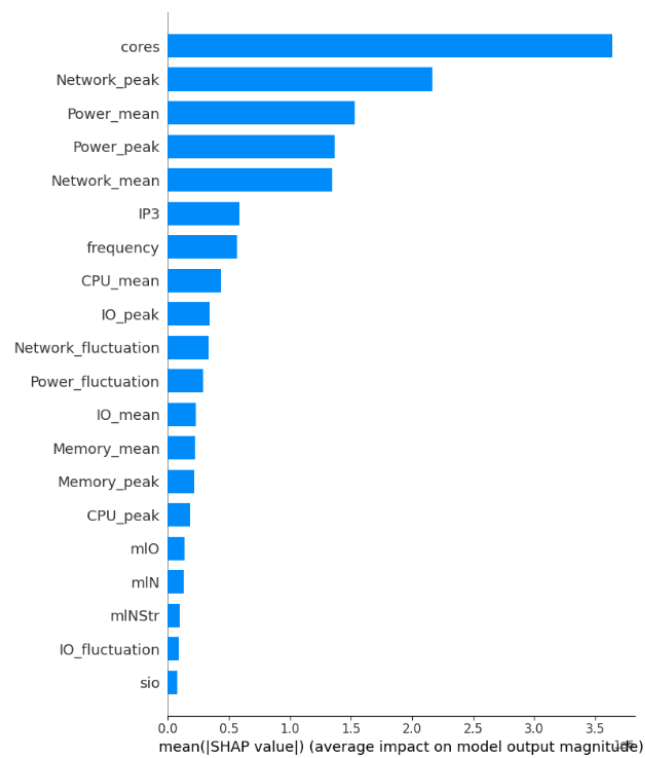


Figure 61. Mean Absolute SHAP Values for Streaming Tuples Metric

## Conclusion

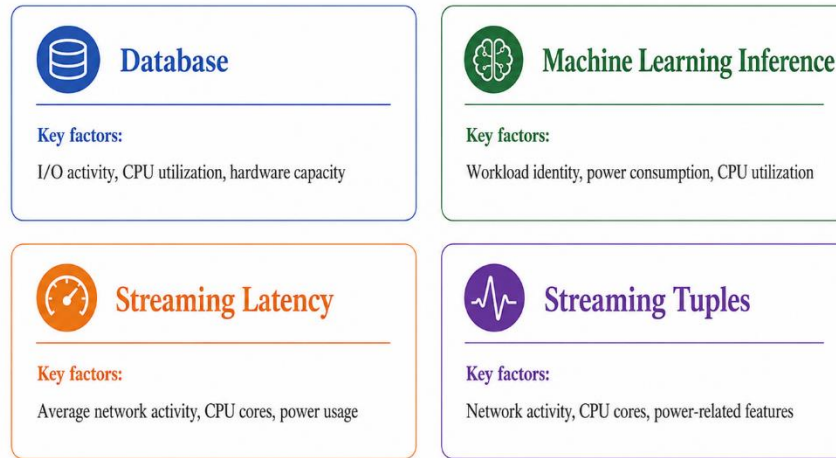
For the streaming tuples application metric, the results show that throughput is mainly influenced by network behaviour and device capacity. Although permutation importance ranks Network\_peak highest and SHAP ranks cores highest, both methods agree that network activity and CPU core count are central to the prediction. Power-related features and the IP3 workload also contribute, indicating that execution intensity and co-located network stress affect streaming throughput. This confirms that the fingerprint representation captures the key factors affecting streaming tuples performance.

### 4.3.1.5 Summary of Individual Feature Importance

Figure 62 summarizes the main findings from the individual feature importance analysis across the four-application metrics. The results show that different performance targets are influenced by different groups of features. Database performance is mainly driven by input/output activity, while machine learning inference is influenced by a combination of workload identity, power consumption, and CPU utilization. This indicates that inference performance is not determined only by the type of machine learning workload, but also by the resource intensity observed during execution. For the streaming metrics, network-related features are the most important, with streaming latency mainly affected by average network activity and streaming tuples influenced by both network activity and CPU core count. The figure also highlights that co-located workloads have visible effects in some cases, especially for database and streaming workloads, confirming that workload fingerprints capture both resource behaviour and interference-related information.

## Summary of Individual Feature Importance Across Application Metrics

Main predictive factors identified by permutation importance and SHAP analysis



Concise comparison of the main factors influencing performance across the four application metrics.

Figure 62. Summary of Individual Feature Importance

### 4.3.2 Grouped Feature Importance

After analysing the influence of individual features, the evaluation is extended to grouped feature importance to examine the impact of broader resource categories. This analysis helps determine whether each application metric is mainly influenced by CPU, memory, input/output, network, power, or workload identity features, rather than focusing only on single variables. As shown in Figure 63, the same feature grouping introduced in Chapter 3 is repeated here for easier reference while discussing the results. For each application metric, grouped permutation importance and grouped SHAP analysis are applied to the best-performing model, allowing the results to be interpreted both in terms of prediction sensitivity and overall group-level contribution.

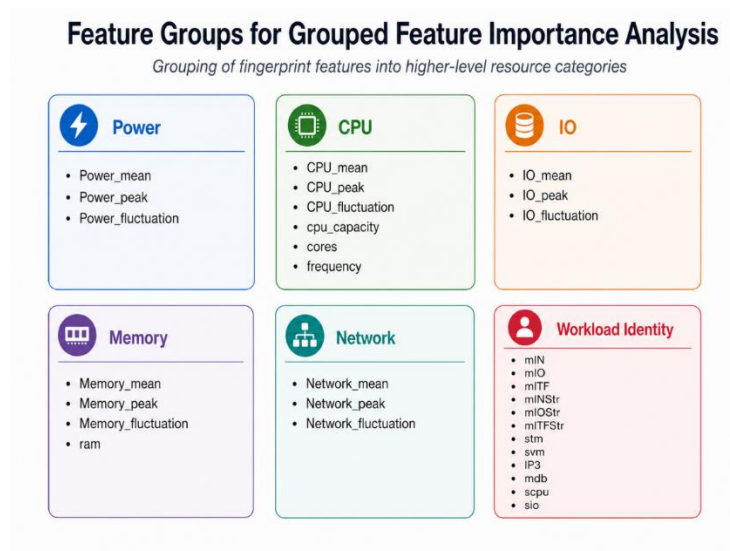


Figure 63. Feature Groups

#### 4.3.2.1 Grouped Feature Importance for Database Application Metric

##### Grouped Permutation Importance

Figure 64 presents the grouped permutation importance results for the database application metric, computed using the Extra Trees model. Unlike the individual analysis, which examines each feature separately, this analysis evaluates related features together as resource groups.

The most influential group is CPU, with a grouped importance value of approximately 0.530. This shows that the combined CPU-related features have the strongest group-level effect on database performance prediction. This includes CPU utilization, CPU capacity, number of cores, and processor frequency.

The second most important group is Input/Output, with an importance value of approximately 0.456. This is consistent with the individual feature importance results, where IO\_mean was the strongest single predictor.

The remaining groups have much smaller effects. Memory and Workload Identity contribute only slightly, with importance values of approximately 0.020 and 0.012,

respectively. Power and Network are close to zero, indicating that they have little influence on the database prediction at the group level.

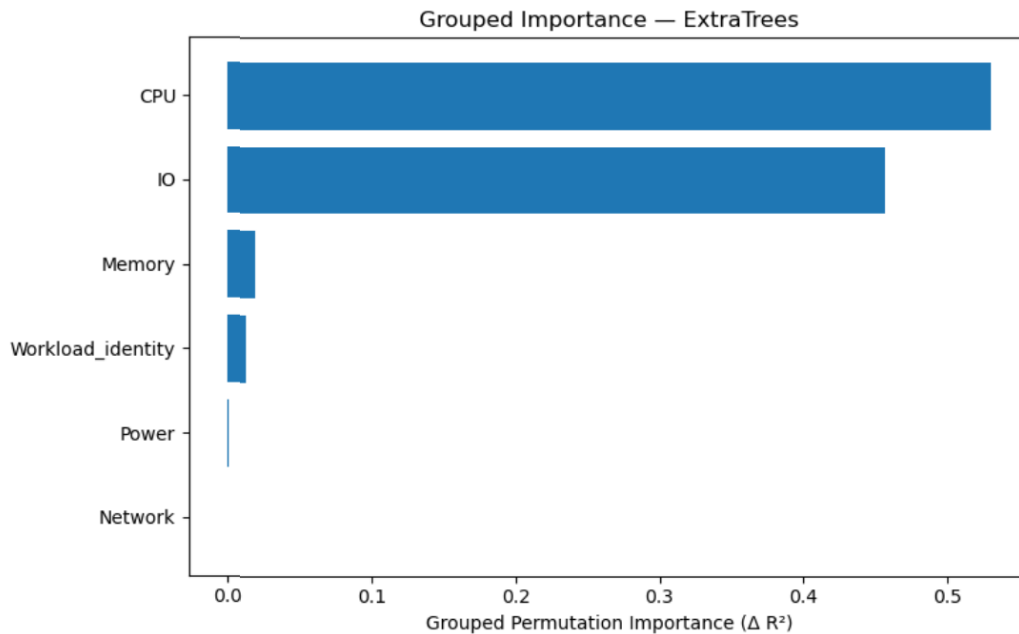


Figure 64. Grouped Permutation Importance for Database Application Metric

### Grouped SHAP Importance

Figure 65 presents the grouped SHAP importance results for the database application metric, computed using the Extra Trees model. In this analysis, the SHAP values of all features within the same group are summed and then averaged across the test samples, producing one importance score for each resource category.

The most influential group is Input/Output, with a mean grouped SHAP value of approximately 813.63. This shows that input/output features have the strongest overall contribution to the database performance predictions when their SHAP effects are considered together. This result aligns with the individual SHAP analysis, where IO\_mean was the dominant feature.

The second most important group is CPU, with a mean grouped SHAP value of approximately 476.39. Although CPU was ranked slightly higher than Input/Output in the grouped permutation importance results, the grouped SHAP analysis places

Input/Output first. This difference indicates that both groups are important, but SHAP attributes the larger average prediction contribution to input/output behaviour.

The remaining groups have much smaller contributions. Memory has a grouped SHAP value of approximately 87.10, followed by Power with 45.40 and Workload Identity with 33.90. These groups provide some additional information, but their influence is clearly lower than that of Input/Output and CPU. Network has the smallest contribution, with a grouped SHAP value of approximately 13.90, confirming that network-related features have limited relevance for database performance prediction in this dataset.

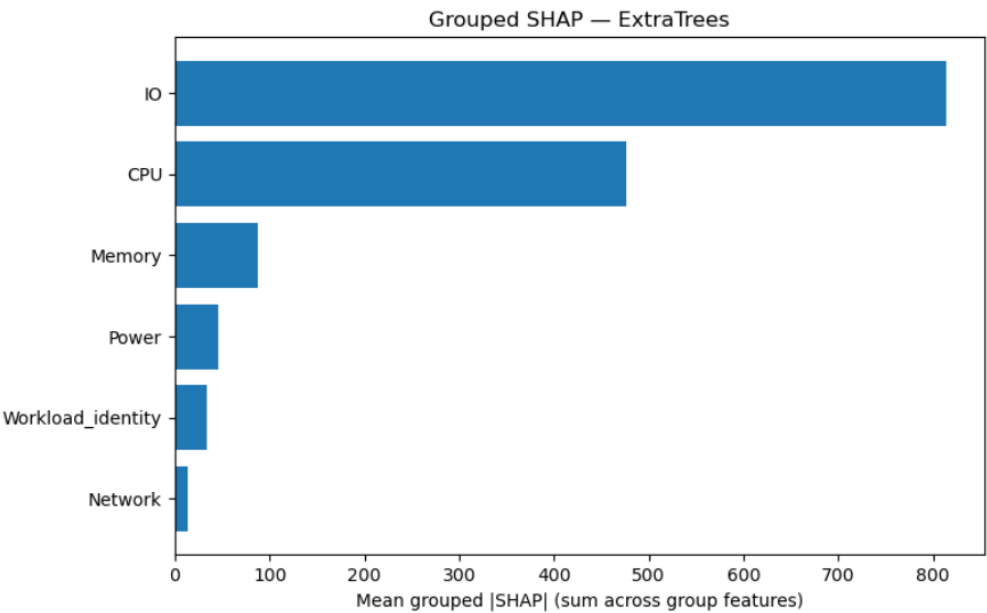


Figure 65. Grouped SHAP Importance for Database Application Metric

### Conclusion

The grouped feature importance results for the database application metric show that database performance is mainly influenced by Input/Output and CPU features. Grouped permutation importance ranks CPU slightly higher, while grouped SHAP ranks Input/Output as the strongest group. Despite this difference, both methods agree that these two resource categories dominate the prediction. Memory, Power, Workload Identity, and Network have much smaller contributions, indicating that they provide only limited additional information for database performance prediction.

#### 4.3.2.2 Grouped Feature Importance for the Machine Learning Application Metric

##### Grouped Permutation Importance

Figure 66 presents the grouped permutation importance results for the machine learning application metric, computed using the Extra Trees model. The results show that Power is the most influential group, with a grouped importance value of approximately 0.693. This indicates that power-related features have the strongest group-level effect on the model's prediction of machine learning inference performance.

The next two most important groups are Workload Identity and CPU, with importance values of approximately 0.452 and 0.446, respectively. Their very similar values show that both the type of machine learning workload being executed and the CPU-related resource features provide substantial predictive information. This is consistent with the individual feature importance results, where workload identity features such as mlO and mlTF, together with CPU\_mean and CPU\_peak, appeared among the most influential individual predictors.

The remaining groups have much smaller effects. Memory contributes slightly, with an importance value of approximately 0.011, while Input/Output and Network are almost zero. This indicates that, at the group level, machine learning inference performance is mainly explained by power consumption, workload identity, and CPU-related features, while memory, input/output, and network behaviour provide limited additional information.

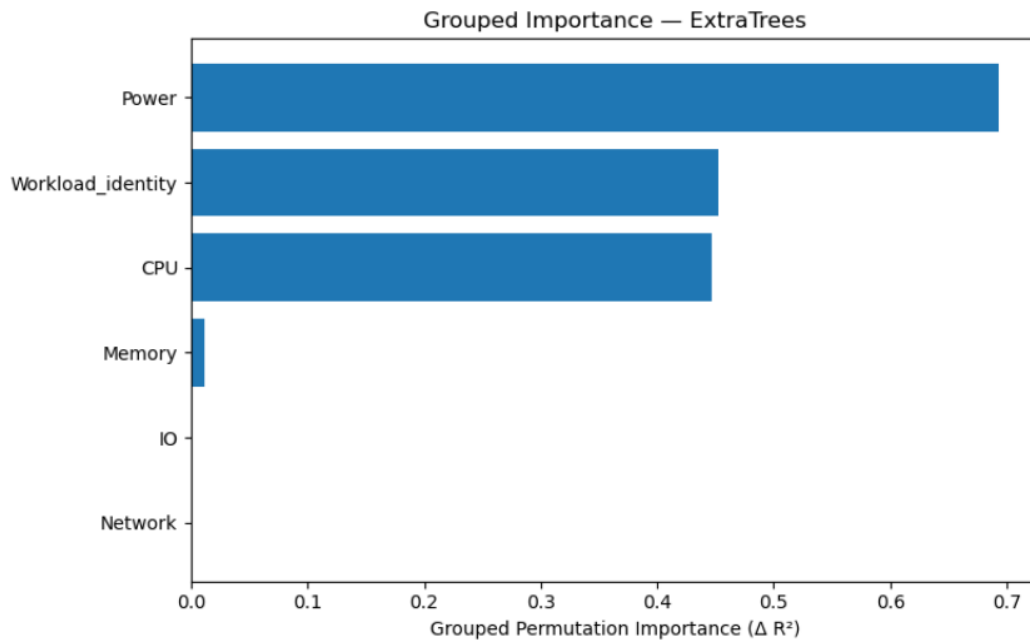


Figure 66. Grouped Permutation Importance for Machine Learning Application Metric

### Grouped SHAP Importance

Figure 67 presents the grouped SHAP importance results for the machine learning application metric, computed using the Extra Trees model. The results show that Power is again the most influential group, with a mean grouped SHAP value of approximately 1402.57. This confirms that power-related features make the largest group-level contribution to the model's machine learning inference predictions.

The second most important group is CPU, with a mean grouped SHAP value of approximately 629.56. This shows that CPU-related features also play a major role in explaining the predictions. Workload Identity follows with a value of approximately 463.59, indicating that the type of inference workload remains an important factor, although grouped SHAP ranks it slightly below CPU.

The remaining groups have much smaller contributions. Memory reaches approximately 153.30, while Input/Output and Network have values of approximately 40.23 and 35.27, respectively. This indicates that memory, input/output, and network features provide only limited additional information compared with Power, CPU, and Workload Identity.

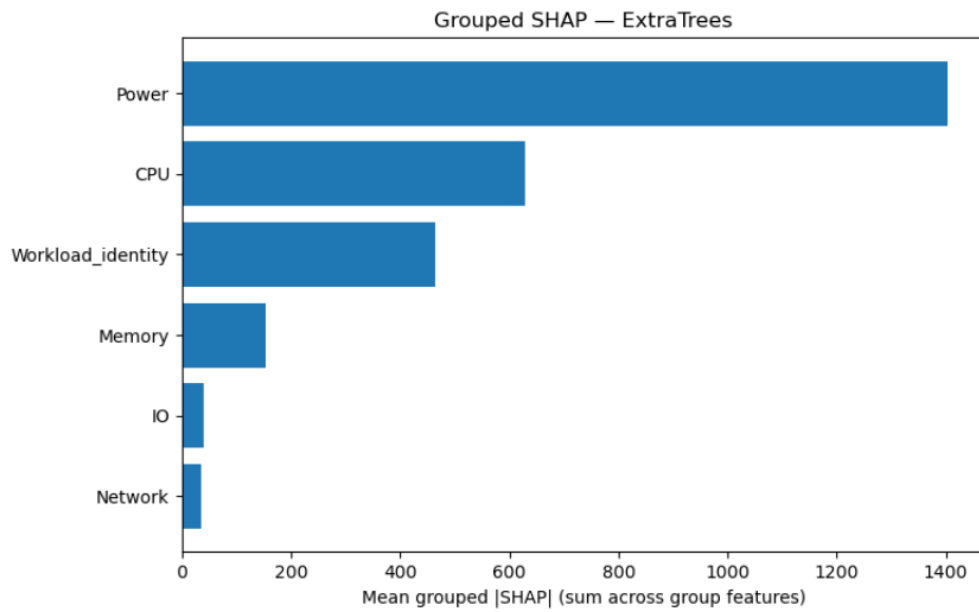


Figure 67. Grouped SHAP Importance for Machine Learning Application Metric

## Conclusion

The grouped feature importance results for the machine learning application metric show that Power is the dominant resource group in both grouped permutation importance and grouped SHAP analysis. CPU and Workload Identity also have strong influence, although their order differs slightly between the two methods. This indicates that machine learning inference performance is mainly explained by execution intensity, processor usage, and the type of inference workload being executed. In contrast, Memory, Input/Output, and Network contribute much less, suggesting that they provide only limited additional information for this prediction task.

### 4.3.2.3 Grouped Feature Importance for the Streaming Latency Application Metric

#### Grouped Permutation Importance

Figure 68 presents the grouped permutation importance results for the streaming latency application metric, computed using the Decision Tree model. The results show that

Network is the dominant group by a large margin, with a grouped importance value of approximately 1.688. This indicates that network-related features have the strongest group-level effect on the prediction of streaming latency.

The next most important groups are Power and CPU, with importance values of approximately 0.265 and 0.223, respectively. These results show that execution intensity and device capacity also contribute to latency prediction, although their influence is much smaller than that of network behaviour.

Memory has a smaller contribution, with an importance value of approximately 0.072, while Workload Identity contributes only slightly, with a value of approximately 0.036. Input/Output has the lowest importance, at approximately 0.008, indicating that input/output activity has little influence on streaming latency at the group level.

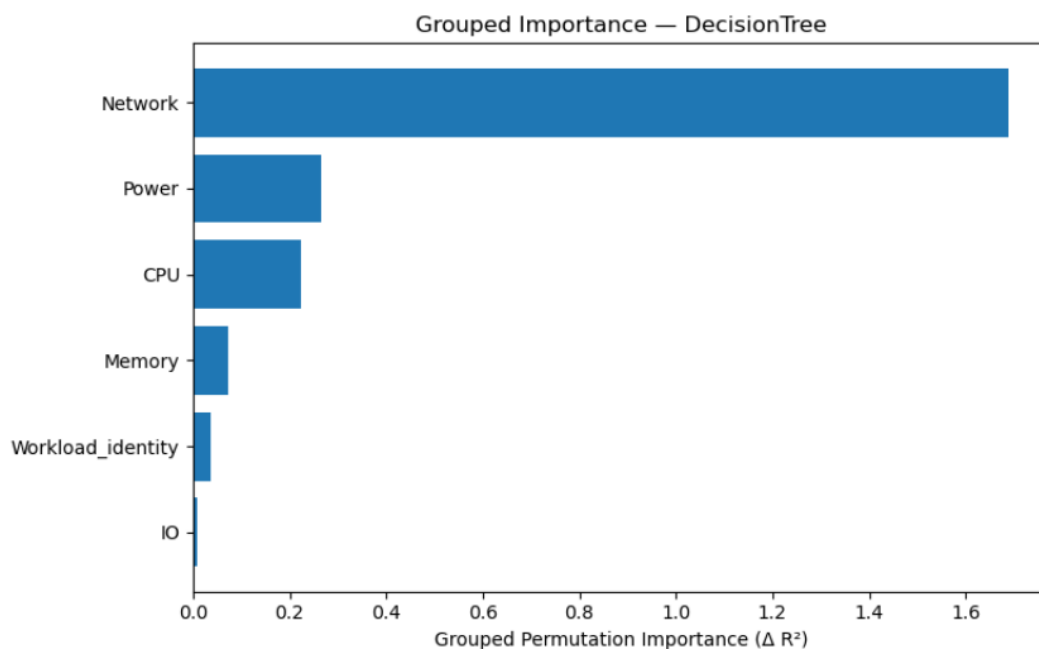


Figure 68. Grouped Permutation Importance for Streaming Latency Application Metric

## Grouped SHAP Importances

Figure 69 presents the grouped SHAP importance results for the streaming latency application metric, computed using the Decision Tree model. The results again show Network as the dominant group, with a mean grouped SHAP value of approximately 475.84. This confirms that network-related features make the largest group-level contribution to the streaming latency predictions.

The next most important group is CPU, with a mean grouped SHAP value of approximately 113.39, followed by Power with approximately 92.45. This shows that device capacity and execution intensity also contribute to the prediction, but their influence is much smaller than that of network behaviour.

The remaining groups have lower contributions. Memory and Input/Output have mean grouped SHAP values of approximately 40.69 and 37.67, respectively, while Workload Identity has the smallest value, approximately 11.12. This indicates that these groups provide only limited additional information compared with Network, CPU, and Power.

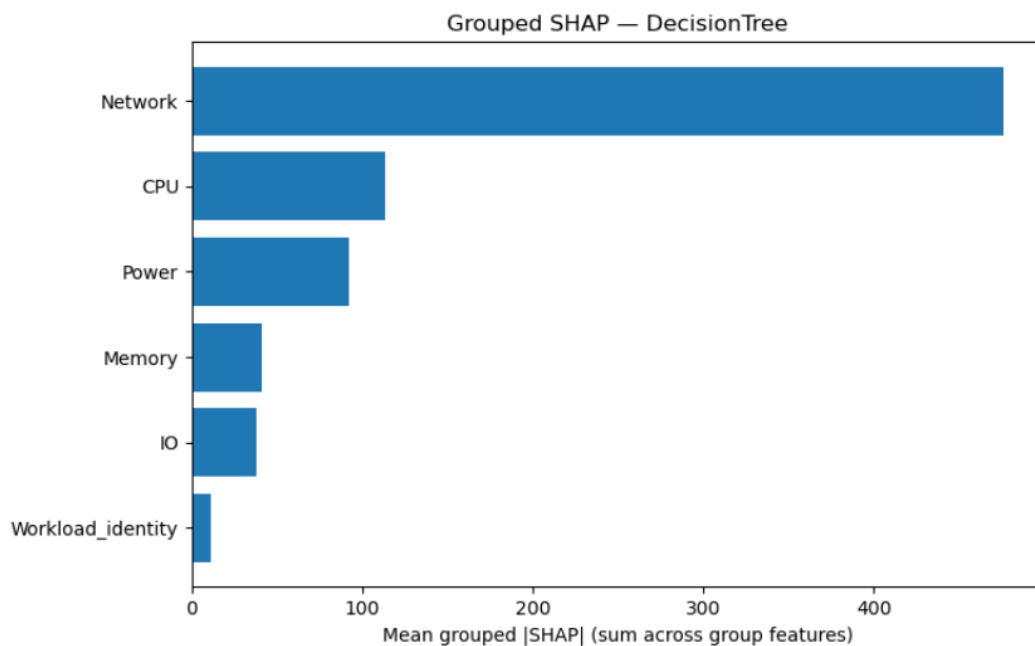


Figure 69. Grouped SHAP Importance for Streaming Latency Application Metric

## **Conclusion**

The grouped feature importance results for the streaming latency application metric show that Network is the dominant feature group in both grouped permutation importance and grouped SHAP analysis. CPU and Power also contribute to the prediction, but their influence is much smaller compared with network-related features. Memory, Input/Output, and Workload Identity have limited group-level importance. This confirms that streaming latency is mainly explained by network behaviour, with additional but secondary influence from device capacity and execution intensity.

### **4.3.2.4 Grouped Feature Importance for the Streaming Tuples Application Metric**

#### **Grouped Permutation Importance**

Figure 70 presents the grouped permutation importance results for the streaming tuples application metric, computed using the XGBoost model. The results show that Network is the most influential group, with a grouped importance value of approximately 0.545. This indicates that network-related features have the strongest group-level effect on the prediction of streaming tuple throughput.

The second most important group is CPU, with an importance value of approximately 0.291, showing that device capacity and CPU-related behaviour also play an important role in predicting the number of processed tuples. Power follows with an importance value of approximately 0.231, suggesting that execution intensity provides additional predictive information.

The remaining groups have smaller effects. Workload Identity contributes moderately, with an importance value of approximately 0.081, indicating that the presence of specific co-located workloads still affects streaming throughput. Input/Output and Memory have lower importance values of approximately 0.041 and 0.022, respectively, showing that they provide limited additional information compared with Network, CPU, and Power.

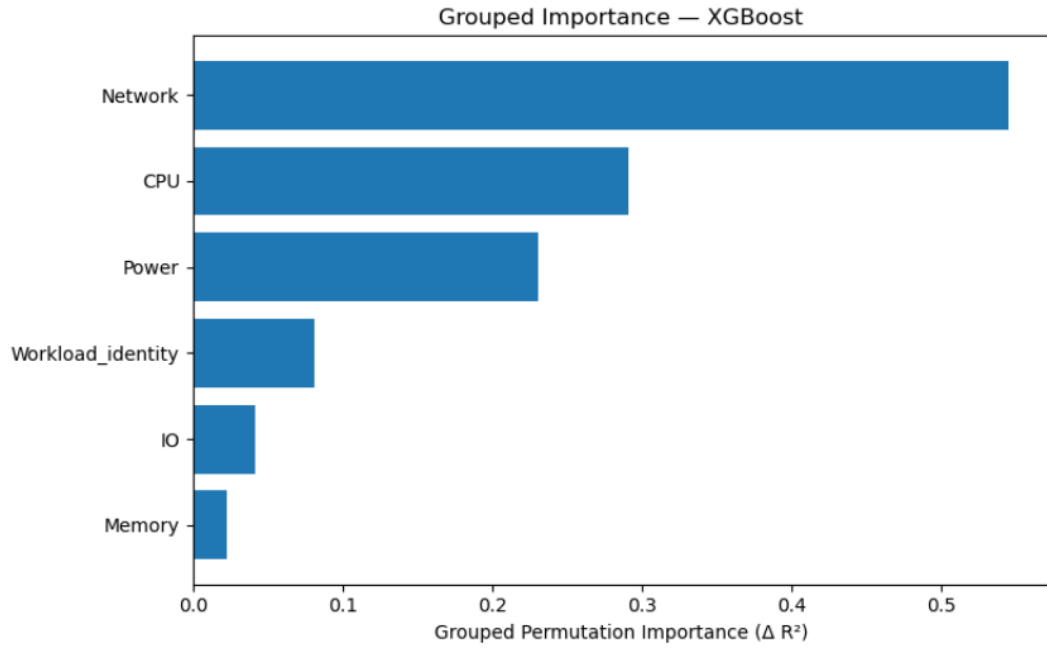


Figure 70. Grouped Permutation Importance for Streaming Tuples Application Metric

### Grouped SHAP Importance

Figure 71 presents the grouped SHAP importance results for the streaming tuples application metric, computed using the XGBoost model. The results show that Network is again the most influential group, with a mean grouped SHAP value of approximately  $1.39 \times 10^6$ . This confirms that network-related features make the largest group-level contribution to the prediction of streaming tuple throughput.

The second most important group is Power, with a mean grouped SHAP value of approximately  $1.17 \times 10^6$ , followed by CPU with approximately  $8.66 \times 10^5$ . This shows that execution intensity and CPU-related features also play a strong role in explaining the model predictions, although Network remains the dominant group.

The remaining groups have noticeably smaller contributions. Input/Output has a mean grouped SHAP value of approximately  $2.43 \times 10^5$ , followed by Memory with approximately  $1.35 \times 10^5$  and Workload Identity with approximately  $1.02 \times 10^5$ . These

groups provide some additional information, but their influence is clearly lower than that of Network, Power, and CPU.

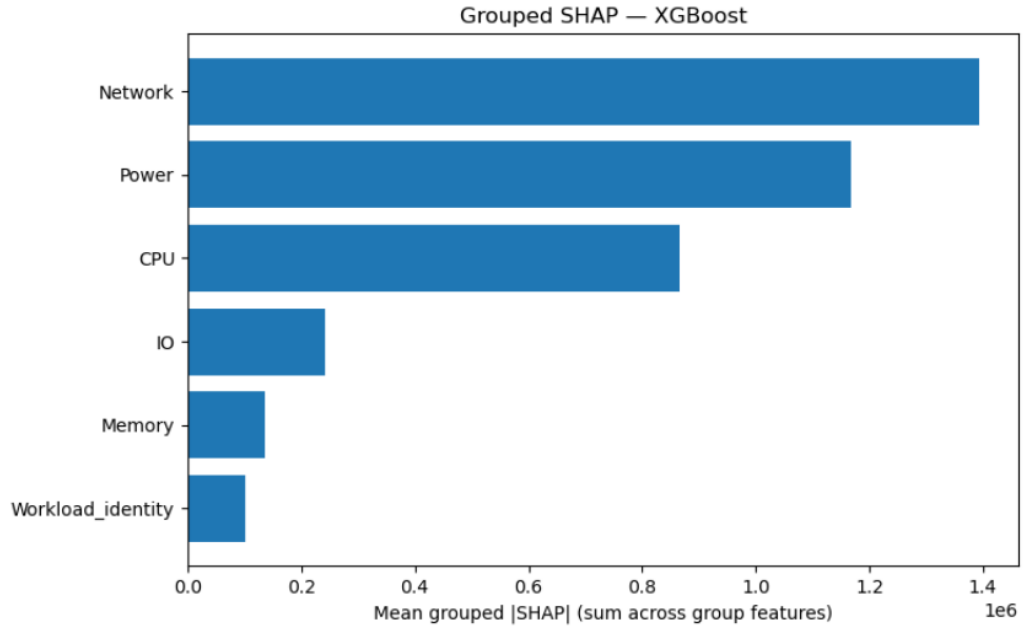


Figure 71. Grouped SHAP Importance for Streaming Tuples Application Metric

## Conclusion

The grouped feature importance results for the streaming tuples application metric show that Network is the dominant feature group in both grouped permutation importance and grouped SHAP analysis. CPU and Power also have strong influence, although grouped permutation importance ranks CPU above Power, while grouped SHAP ranks Power above CPU. This indicates that streaming tuple throughput is mainly explained by network behaviour, device capacity, and execution intensity. Workload Identity, Input/Output, and Memory provide smaller additional contributions, suggesting that they are less influential at the group level for this prediction task.

### 4.3.2.5 Summary of Grouped Feature Importance

Figure 72 summarizes the grouped feature importance findings across the four application metrics and connects them with the individual feature importance results shown earlier in Figure 62. The grouped analysis confirms that the dominant individual features belong to the most influential resource groups. For the database metric, the individual analysis highlighted IO\_mean as the strongest feature, while the grouped analysis shows that

Input/Output and CPU are the dominant resource groups. For machine learning inference, the individual importance results emphasized workload identity, power consumption, and CPU utilization, and the grouped results similarly identify Power, CPU, and Workload Identity as the main groups.

For the two streaming metrics, both individual and grouped analyses show the central role of network behaviour. In streaming latency, Network\_mean was the strongest individual feature, and the grouped analysis confirms Network as the dominant group. Similarly, for streaming tuples, individual importance highlighted Network\_peak, Network\_mean, and cores, while the grouped results show that Network, CPU, and Power are the main resource groups. Overall, Figure 72 reinforces the findings from Figure 62 by showing that the most important individual predictors are part of broader resource categories that consistently explain performance differences across the application metrics.

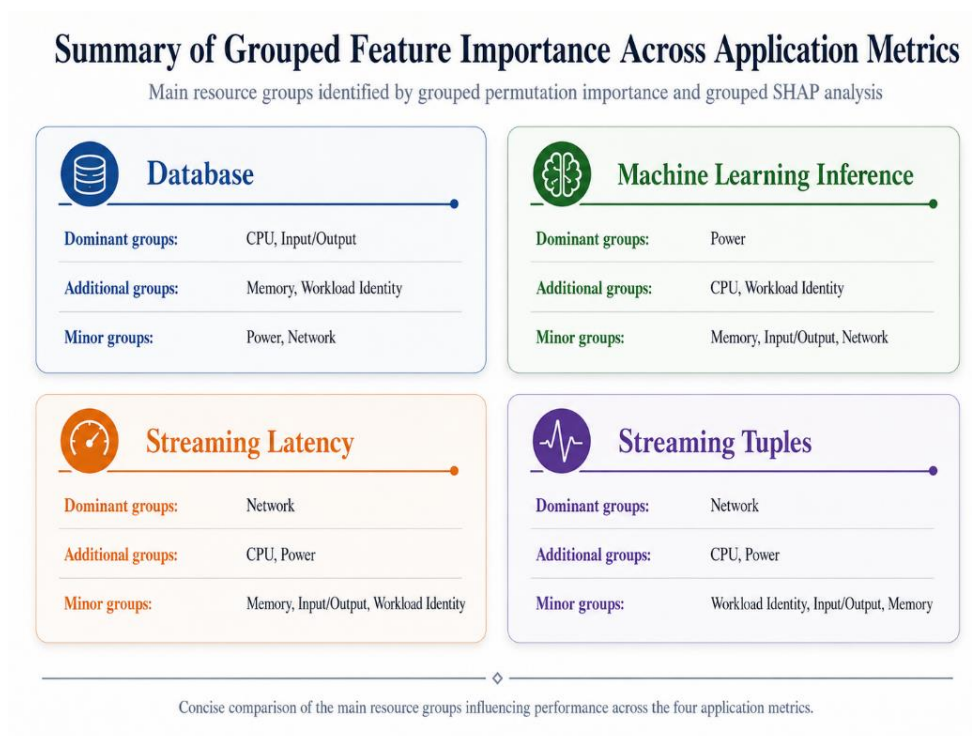


Figure 72. Summary of Grouped Feature Importance

## 4.4 Clustering

After evaluating model performance and feature importance, clustering is applied to explore whether workload executions with similar resource utilization patterns can be grouped together. Unlike regression, clustering is not used to predict the performance value. Instead, it is used as an unsupervised analysis step to identify groups of experiments that behave similarly in terms of CPU, memory, input/output, network, and power usage.

In this thesis, KMeans clustering is used for this analysis. KMeans partitions the dataset into  $k$  clusters by grouping observations according to their similarity, with the objective of forming compact clusters around their centroids [25]. Since KMeans requires the number of clusters,  $k$ , to be selected before fitting the model, diagnostic methods are first used to support this choice.

To select an appropriate value of  $k$ , the elbow method and silhouette analysis are applied. The elbow method examines how the within-cluster variation changes as the number of clusters increases. A suitable value of  $k$  is usually selected near the point where adding more clusters leads to only limited improvement, forming an “elbow” in the curve [31]. Silhouette analysis complements this by measuring how well each sample fits within its assigned cluster compared with neighbouring clusters. Higher silhouette values generally indicate better cluster separation and cohesion [32].

For each application-specific dataset, elbow and silhouette plots are generated across a range of candidate  $k$  values. After selecting the number of clusters for each dataset, KMeans is applied to the standardized feature matrix. Standardization is required because the resource utilization features have different numerical scales, and without scaling, features with larger values could dominate the clustering process.

After clustering, the centroids are transformed back to the original feature scale to make the results easier to interpret. The centroid matrix summarizes the average resource utilization pattern of each cluster and is used to characterize the main workload behaviour

patterns observed in the database, machine learning inference, streaming latency, and streaming tuples datasets.

#### 4.4.1 Cluster Selection Using Elbow and Silhouette Analysis

As discussed above, KMeans requires the number of clusters,  $k$ , to be selected before the model is fitted. To support this selection, the elbow method and silhouette analysis are applied to each application-specific dataset. The elbow method shows how inertia changes as  $k$  increases, while the silhouette score provides an indication of how well-separated and coherent the resulting clusters are. By considering both plots together, an appropriate value of  $k$  is selected for each dataset before applying KMeans clustering.

##### 4.4.1.1 Cluster Selection Using Elbow and Silhouette Analysis for Database Application Metric

From Figure 73, the selected number of clusters for the database application metric is  $k = 8$ . In the elbow plot, the inertia decreases rapidly for smaller values of  $k$ , but the improvement becomes much smaller after approximately  $k = 8$ . This indicates that adding more clusters beyond this point provides only limited additional benefit. At the same time, the silhouette plot reaches its highest value at  $k = 8$ , showing that this value produces the best cluster separation among the tested options. Therefore,  $k = 8$  is selected as the most suitable number of clusters for the database dataset.

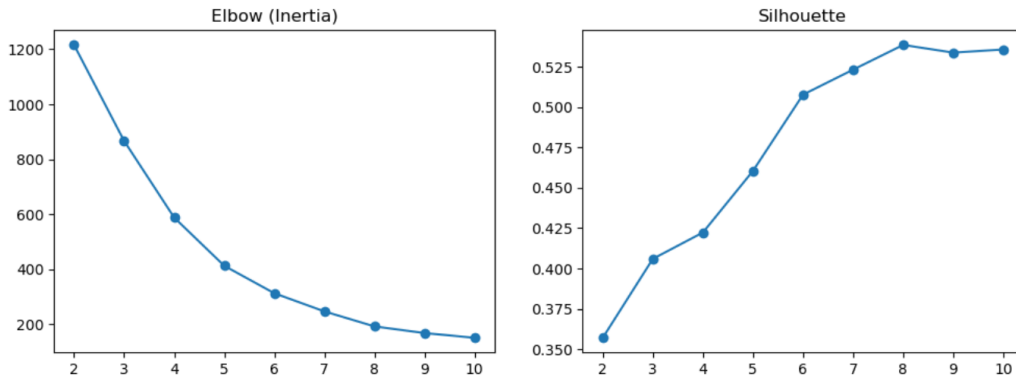


Figure 73. Elbow and Silhouette Analysis for Selecting  $k$  in the Database Application Metric

#### 4.4.1.2 Cluster Selection Using Elbow and Silhouette Analysis for Machine Learning Application Metric

Figure 74 presents the elbow and silhouette analysis for the machine learning application metric. Based on these diagnostics, the selected number of clusters is  $k = 9$ . In the elbow plot, inertia continues to decrease as  $k$  increases, but after  $k = 9$  the improvement becomes very small, indicating that adding more clusters provides limited additional benefit. The silhouette plot also supports this choice, as the score reaches its highest value at  $k = 9$  and remains almost unchanged at  $k = 10$ . Therefore,  $k = 9$  is selected as the most suitable number of clusters for the machine learning dataset.

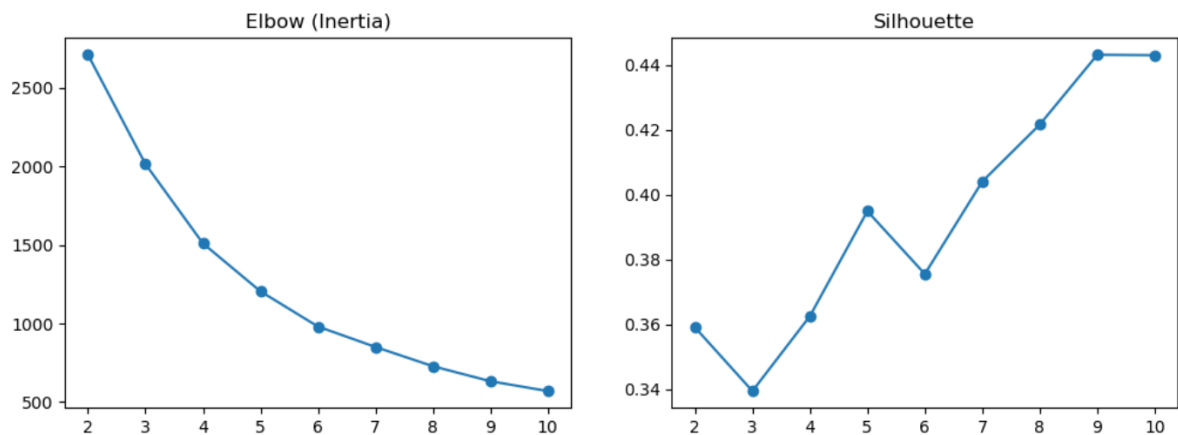


Figure 74. Elbow and Silhouette Analysis for Selecting  $k$  in Machine Learning Application Metric

#### 4.4.1.3 Cluster Selection Using Elbow and Silhouette Analysis for Streaming Latency Application Metric

Figure 75 presents the elbow and silhouette analysis for the streaming latency application metric. Based on these diagnostics, the selected number of clusters is  $k = 8$ . In the elbow plot, inertia decreases rapidly for smaller values of  $k$ , while the improvement becomes more gradual after approximately  $k = 8$ . The silhouette plot also supports this choice, as the score is high at  $k = 8$  and remains within the strongest range of the tested values.

Although  $k = 10$  gives a slightly higher silhouette score,  $k = 8$  is selected because it provides a good balance between cluster quality and model simplicity.

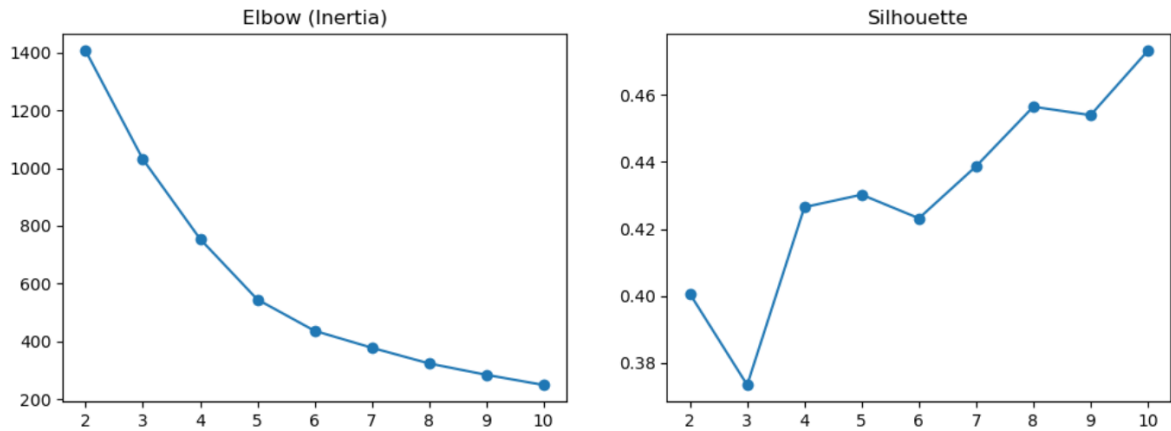


Figure 75. Elbow and Silhouette Analysis for Selecting  $k$  in Streaming Latency Application Metric

#### 4.4.1.4 Cluster Selection Using Elbow and Silhouette Analysis for Streaming Tuples Application Metric

Figure 76 presents the elbow and silhouette analysis for the streaming tuples application metric. Based on these diagnostics, the selected number of clusters is  $k = 8$ . In the elbow plot, inertia decreases strongly for smaller values of  $k$ , but the improvement becomes more gradual after approximately  $k = 8$ . The silhouette plot also shows a strong score at  $k = 8$ , indicating good cluster separation. Although  $k = 10$  achieves a slightly higher silhouette score,  $k = 8$  is selected because it provides a good balance between clustering quality and model simplicity.

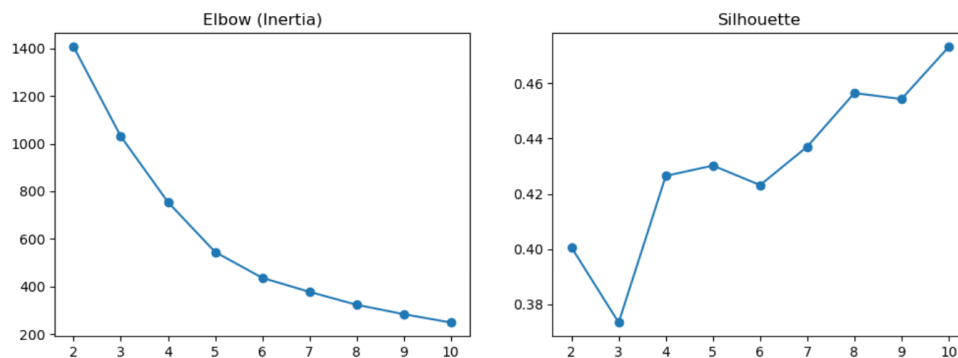


Figure 76. Elbow and Silhouette Analysis for Selecting  $k$  in Streaming Tuples Application Metric

#### 4.4.2 KMeans Clustering and Centroid Interpretation

After selecting the number of clusters using the elbow and silhouette analysis, KMeans clustering is applied separately to each application-specific dataset. Figure 77 summarizes the selected  $k$  values used for the clustering process. The database dataset is clustered using  $k = 8$ , the machine learning inference dataset is clustered using  $k = 9$ , and both the streaming latency and streaming tuples datasets are clustered using  $k = 8$ .

For each dataset, clustering is performed on the standardized resource utilization features, so that differences in feature scale do not dominate the results. After fitting the KMeans model, the cluster centroids are transformed back to the original feature scale to make them easier to interpret. Each centroid represents the average resource utilization profile of one cluster, allowing the analysis to identify different workload behaviour patterns across the application metrics.

| Application Metric         | Selected $k$ |
|----------------------------|--------------|
| Database                   | 8            |
| Machine Learning Inference | 9            |
| Streaming Latency          | 8            |
| Streaming Tuples           | 8            |

*Figure 77. Summary of the selected  $k$  values used for KMeans clustering across the four application metrics*

##### 4.4.2.1 Centroid Analysis for Database Application Metric

Table 2 presents the centroid matrix for the database application metric using  $k = 8$ , while Figure 78 shows the corresponding normalized centroid heatmap. The table reports the original average feature values for each cluster, whereas the heatmap normalizes each feature column between 0 and 1 to make resource patterns easier to compare visually.

The database executions are mainly separated by input/output activity, CPU utilization, memory usage, network behaviour, and power consumption. Clusters 0, 4, and 7 show high input/output peak values, indicating executions with strong disk activity. CPU utilization also distinguishes several clusters, with clusters 1, 3, and 4 showing very high CPU\_mean values. Among these, cluster 4 is the most resource-intensive case, combining high CPU usage with the highest memory consumption and high power draw.

A clear outlier appears in cluster 5, which has much higher Network\_peak and Network\_mean values than the other clusters. This suggests that some database executions are associated with strong network activity, possibly due to communication-heavy conditions or co-located workloads. In contrast, cluster 3 has relatively low input/output and power values, representing a lighter profile despite its high CPU utilization.

Overall, Table 2 and Figure 78 show that database executions form distinct resource-usage profiles, including input/output-intensive, CPU-intensive, high-memory/high-power, and network-heavy clusters. This supports clustering as a complementary method for characterizing workload behaviour beyond regression and feature importance.

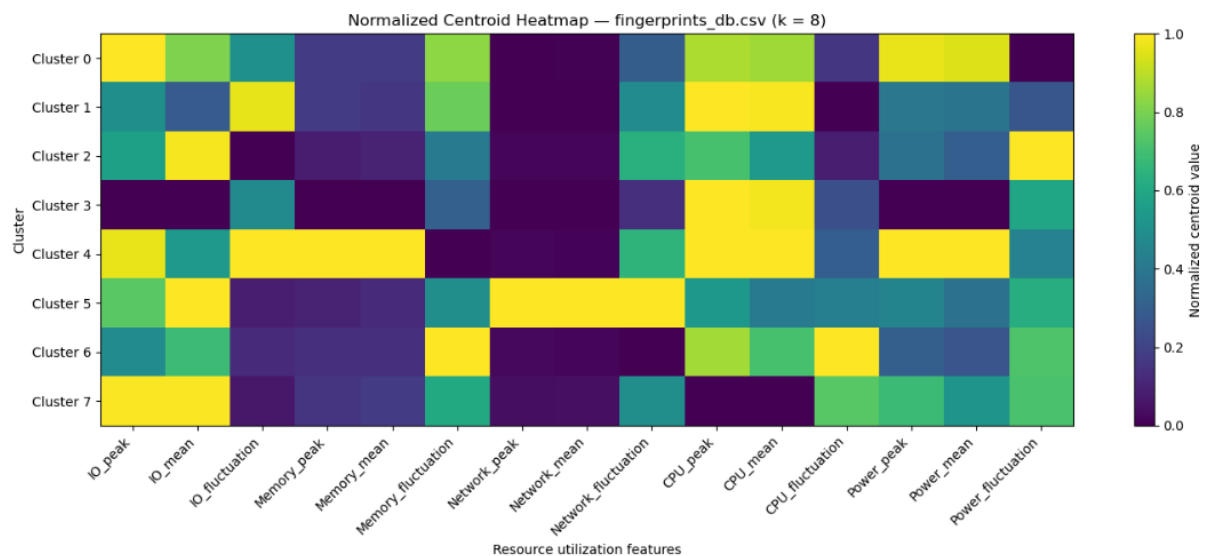


Figure 78. Normalized KMeans Centroid Heatmap for the Database Application Metric

| cluster | IO_peak       | IO_mean      | IO_fluctuation | Memory_peak  | Memory_mean  | Memory_fluctuation | Network_peak | Network_mean  | Network_fluctuation | CPU_peak  | CPU_mean  | CPU_fluctuation | Power_peak | Power_mean | Power_fluctuation |
|---------|---------------|--------------|----------------|--------------|--------------|--------------------|--------------|---------------|---------------------|-----------|-----------|-----------------|------------|------------|-------------------|
| 0       | 199344.597143 | 18995.181886 | -5.000000e-01  | 9193.748429  | 7562.914429  | 0.642857           | 2.064071e+04 | 7894.602764   | 0.071429            | 92.244500 | 86.759771 | -0.428571       | 407.714286 | 377.131714 | -0.642857         |
| 1       | 112759.180148 | 9790.557681  | -3.703704e-02  | 9269.972400  | 7115.565130  | 0.592593           | 1.756290e+04 | 4995.482433   | 0.296296            | 99.800719 | 96.378922 | -0.629630       | 170.888889 | 156.295526 | -0.333333         |
| 2       | 124644.974783 | 22194.796578 | -1.000000e+00  | 5998.008930  | 5309.308957  | 0.260870           | 3.541737e+04 | 13244.654043  | 0.478261            | 81.861635 | 63.997887 | -0.521739       | 162.913043 | 123.481096 | 0.521739          |
| 3       | 26919.041341  | 4616.901689  | -5.227273e-01  | 3256.569525  | 2420.994355  | 0.159091           | 1.926134e+04 | 6663.820270   | -0.113636           | 99.990686 | 95.870423 | -0.318182       | 7.028932   | 6.424705   | 0.045455          |
| 4       | 193289.512250 | 14229.964088 | -1.110223e-16  | 37338.544000 | 32147.655887 | -0.125000          | 3.043390e+04 | 8847.931500   | 0.500000            | 99.914350 | 97.053275 | -0.250000       | 421.875000 | 397.809425 | -0.125000         |
| 5       | 155538.357167 | 23442.958025 | -9.166667e-01  | 6651.792267  | 5975.572283  | 0.333333           | 1.035269e+06 | 462885.267446 | 0.916667            | 70.851533 | 54.758804 | -0.083333       | 195.000000 | 151.861050 | 0.083333          |
| 6       | 110169.169748 | 16827.439084 | -8.800000e-01  | 7870.231300  | 6568.698668  | 0.800000           | 4.049436e+04 | 13894.807112  | -0.280000           | 91.411708 | 76.071048 | 0.640000        | 133.360000 | 109.138564 | 0.200000          |
| 7       | 198616.675000 | 22326.837456 | -9.375000e-01  | 8509.365863  | 7642.727356  | 0.437500           | 5.372359e+04 | 24317.295731  | 0.312500            | 37.551106 | 25.274556 | 0.312500        | 290.187500 | 209.278437 | 0.187500          |

Table 2. Centroid Matrix for the Database Application Metric

#### 4.4.2.2 Centroid Analysis for Machine Learning Application Metric

Table 3 presents the KMeans cluster centroids for the machine learning application metric using  $k = 9$ , while Figure 79 shows the corresponding normalized centroid heatmap. The table provides the original centroid values, while the heatmap highlights relative differences between clusters across features with different scales.

The machine learning executions are separated by different combinations of CPU utilization, input/output activity, network behaviour, memory usage, and power consumption. Clusters 0, 2, 3, 4, and 7 show high CPU\_mean values, indicating processor-intensive execution profiles. Cluster 4 is the most resource-intensive cluster, combining very high CPU utilization with the highest memory and power values.

Input/output activity varies across clusters, with clusters 2, 5, and 8 showing the highest IO\_mean values. This may be related to data movement, model loading, or co-located workload activity during inference execution. Network activity forms another distinctive pattern, as clusters 5 and 6 show much higher Network\_peak and Network\_mean values than the remaining clusters, suggesting stronger communication activity or network-related co-location effects.

Power consumption generally follows execution intensity, but it also separates clusters with similar CPU usage. For example, cluster 4 combines high CPU and high power values, while clusters 3 and 7 show high CPU utilization with much lower power consumption. This indicates that similar processor usage can correspond to different resource profiles depending on device characteristics and runtime conditions.

Overall, Table 3 and Figure 79 show that machine learning inference has a diverse centroid structure, including CPU-intensive, high-input/output, network-heavy, and high-memory/high-power clusters. This is consistent with the feature importance results, where workload identity, power, and CPU features were all highly influential.

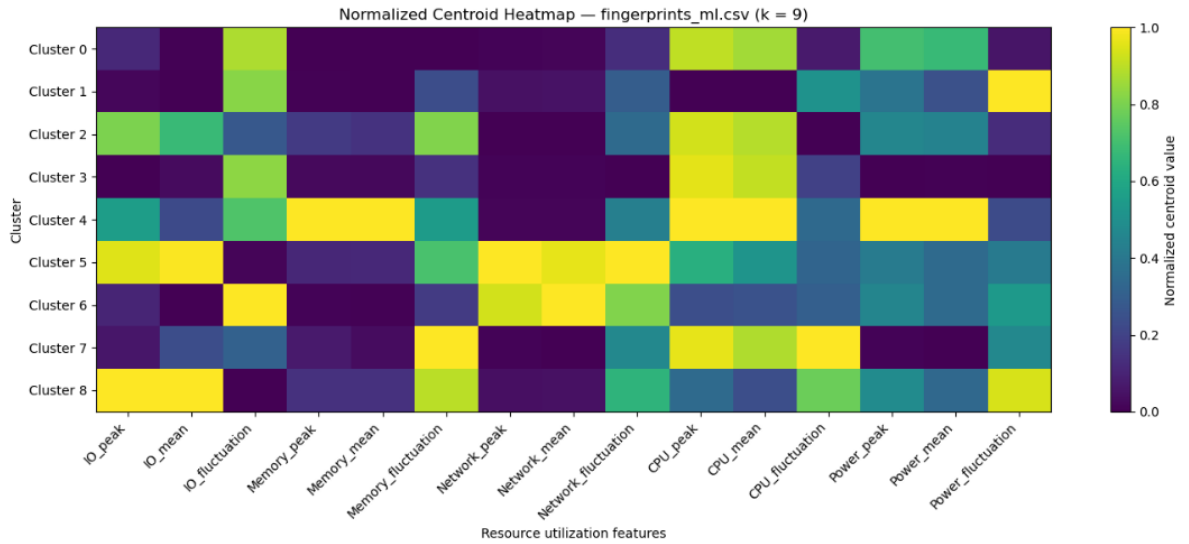


Figure 79. Normalized KMeans Centroid Heatmap for the Machine Learning Application Metric

|         | IO_peak       | IO_mean      | IO_fluctuation | Memory_peak  | Memory_mean  | Memory_fluctuation | Network_peak | Network_mean  | Network_fluctuation | CPU_peak  | CPU_mean  | CPU_fluctuation | Power_peak | Power_mean | Power_fluctuation |
|---------|---------------|--------------|----------------|--------------|--------------|--------------------|--------------|---------------|---------------------|-----------|-----------|-----------------|------------|------------|-------------------|
| cluster |               |              |                |              |              |                    |              |               |                     |           |           |                 |            |            |                   |
| 0       | 46348.122400  | 798.932361   | 0.707317       | 3202.800124  | 2983.102963  | -0.853659          | 3.093091e+04 | 13125.051583  | -0.097561           | 94.341956 | 87.020761 | -4.634146e-01   | 310.804878 | 280.521005 | -0.414634         |
| 1       | 33896.547469  | 683.189094   | 0.600000       | 3387.158403  | 3030.074911  | -0.457143          | 6.810919e+04 | 30885.129820  | 0.085714            | 41.579043 | 29.414471 | -2.775558e-17   | 192.240114 | 131.183531 | 0.885714          |
| 2       | 135811.635560 | 14901.240732 | -0.440000      | 8482.502908  | 6887.140124  | 0.500000           | 2.210975e+04 | 7551.279252   | 0.140000            | 95.859108 | 88.976386 | -5.400000e-01   | 221.080000 | 199.247488 | -0.320000         |
| 3       | 31545.042227  | 1254.094831  | 0.614458       | 4002.465304  | 3507.895895  | -0.614458          | 3.261529e+04 | 12569.594877  | -0.240964           | 97.519389 | 90.136786 | -3.373494e-01   | 49.089663  | 45.584824  | -0.493976         |
| 4       | 103917.975053 | 5408.429124  | 0.411765       | 34369.982588 | 29633.811724 | 0.058824           | 3.615794e+04 | 14334.268047  | 0.235294            | 99.922400 | 96.370465 | -1.764706e-01   | 422.823529 | 395.667688 | -0.176471         |
| 5       | 155051.737867 | 21564.749620 | -0.933333      | 6715.779667  | 6044.946707  | 0.333333           | 1.060399e+06 | 454578.283753 | 0.866667            | 78.482633 | 63.956947 | -2.000000e-01   | 204.733333 | 165.444967 | 0.066667          |
| 6       | 44828.540894  | 653.423087   | 0.937500       | 3500.388938  | 3182.624800  | -0.562500          | 9.899200e+05 | 472309.404769 | 0.656250            | 55.682903 | 46.203191 | -2.187500e-01   | 219.250000 | 164.505519 | 0.250000          |
| 7       | 39398.114643  | 5632.211391  | -0.370370      | 5438.326619  | 3914.399970  | 0.814815           | 3.251804e+04 | 8516.877837   | 0.277778            | 97.787691 | 88.460220 | 5.185185e-01    | 53.401407  | 43.188928  | 0.148148          |
| 8       | 161359.745040 | 21665.711872 | -0.960000      | 7661.665264  | 6732.418624  | 0.640000           | 6.591410e+04 | 28140.858856  | 0.480000            | 61.739776 | 45.537332 | 2.800000e-01    | 228.800000 | 162.779636 | 0.800000          |

Table 3. Centroid Matrix for the Machine Learning Application Metric

#### 4.4.2.3 Centroid Analysis for Streaming Latency Application Metric

Table 4 presents the KMeans cluster centroids for the streaming latency application metric using  $k = 8$ , while Figure 80 shows the corresponding normalized centroid heatmap. The table reports the original cluster centroids, while the heatmap makes the main resource-usage patterns easier to identify visually.

The streaming latency executions are mainly separated by network activity, CPU utilization, input/output behaviour, memory usage, and power consumption. Cluster 4 forms the clearest network-heavy profile, with much higher `Network_peak` and `Network_mean` values than all other clusters. This agrees with the grouped feature importance results, where Network was the dominant resource group for streaming latency prediction.

CPU utilization also separates several clusters. Clusters 1, 2, 3, 5, and 7 show high `CPU_mean` values, indicating strong processor usage. Cluster 2 is particularly resource-intensive, combining high CPU utilization with the highest memory usage and high power consumption. Input/output activity is most visible in cluster 6, which has the highest `IO_peak` and `IO_mean` values, while cluster 0 represents a lighter profile with lower CPU utilization and moderate network activity.

Overall, Table 4 and Figure 80 show that streaming latency executions form network-heavy, CPU-intensive, high-memory/high-power, and input/output-intensive profiles. These results complement the feature importance analysis by showing that latency-related behaviour is mainly shaped by network activity, while other resource dimensions further distinguish the clusters.

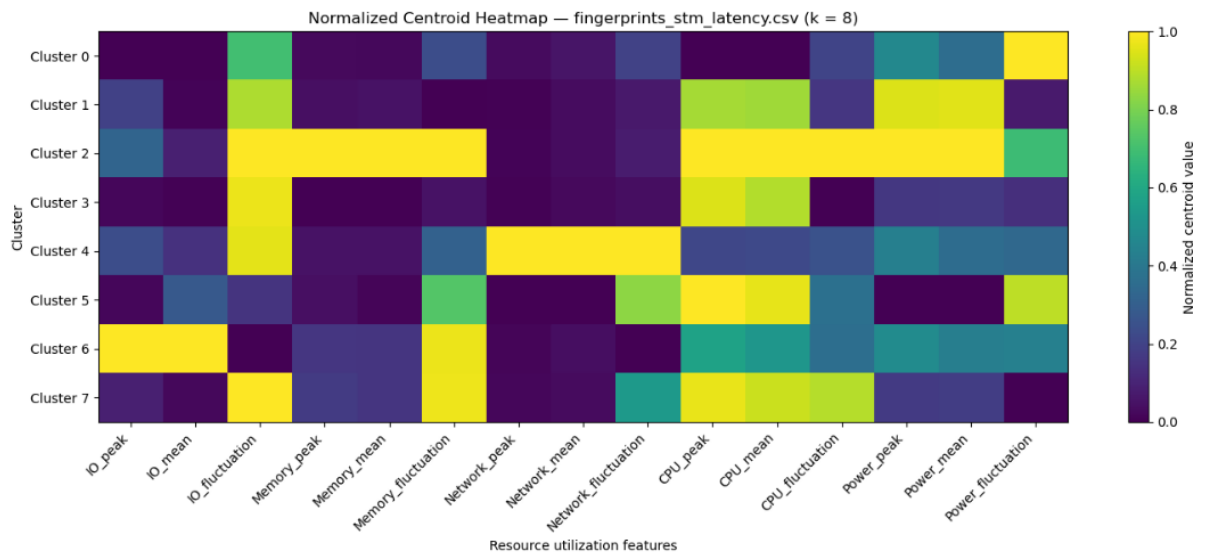


Figure 80. Normalized KMeans Centroid Heatmap for the Streaming Latency Application Metric

| cluster | IO_peak       | IO_mean      | IO_fluctuation | Memory_peak  | Memory_mean  | Memory_fluctuation | Network_peak  | Network_mean  | Network_fluctuation | CPU_peak  | CPU_mean  | CPU_fluctuation | Power_peak | Power_mean | Power_fluctuation |
|---------|---------------|--------------|----------------|--------------|--------------|--------------------|---------------|---------------|---------------------|-----------|-----------|-----------------|------------|------------|-------------------|
| 0       | 28723.054792  | 803.027688   | 0.200000       | 3667.547776  | 3259.465272  | -0.520000          | 68911.831760  | 33050.695640  | -0.440000           | 43.901608 | 28.925796 | 0.200000        | 208.026320 | 148.637812 | 0.520000          |
| 1       | 52301.918855  | 1007.127882  | 0.454545       | 4195.530509  | 3909.085809  | -0.909091          | 46387.529273  | 21183.241473  | -0.636364           | 92.462609 | 84.500327 | 0.181818        | 400.181818 | 373.057745 | -0.272727         |
| 2       | 68579.477625  | 2487.007937  | 0.625000       | 33147.108500 | 28559.314100 | 0.750000           | 49960.130700  | 22270.517712  | -0.625000           | 99.838600 | 93.834800 | 0.500000        | 421.750000 | 390.155538 | 0.250000          |
| 3       | 30944.393371  | 877.402410   | 0.585366       | 2922.746124  | 2686.931910  | -0.829268          | 46055.948107  | 18859.851593  | -0.682927           | 96.891607 | 86.350822 | 0.121951        | 85.518268  | 76.563827  | -0.219512         |
| 4       | 57877.698143  | 3532.439452  | 0.565217       | 4361.386617  | 3914.400952  | -0.391304          | 995682.203478 | 472319.384230 | 0.782609            | 55.728387 | 43.319404 | 0.217391        | 193.173913 | 146.915800 | -0.043478         |
| 5       | 30917.960048  | 6075.032822  | -0.565217      | 4205.292652  | 3022.771543  | 0.304348           | 40457.012687  | 6227.891017   | 0.521739            | 99.925517 | 91.388800 | 0.260870        | 20.421304  | 13.799330  | 0.434783          |
| 6       | 151874.881185 | 19776.123507 | -0.777778      | 7734.831052  | 6673.306296  | 0.703704           | 52373.590459  | 24316.967885  | -0.740741           | 76.104615 | 63.135796 | 0.259259        | 213.851852 | 175.422604 | 0.037037          |
| 7       | 39632.802875  | 1190.473579  | 0.625000       | 8132.780104  | 6657.558692  | 0.708333           | 58511.729225  | 19246.683583  | 0.083333            | 98.075004 | 88.580829 | 0.458333        | 88.852042  | 81.589229  | -0.333333         |

Table 4. Centroid Matrix for the Streaming Latency Application Metric

#### 4.4.2.4 Centroid Analysis for Streaming Tuples Application Metric

Table 5 presents the KMeans cluster centroids for the streaming tuples application metric using  $k = 8$ , while Figure 81 shows the corresponding normalized centroid heatmap. The table provides the original centroid values, while the heatmap highlights the relative differences between clusters.

The streaming tuples dataset is closely related to the streaming latency dataset, since both are derived from the same streaming workload but represent different performance aspects. Streaming latency captures delay, while streaming tuples captures throughput, which explains why several cluster patterns are similar across the two datasets.

The streaming tuples executions are mainly separated by network activity, CPU utilization, input/output behaviour, memory usage, and power consumption. Cluster 5 is the clearest network-heavy profile, with much higher Network\_peak and Network\_mean values than the other clusters. This agrees with the grouped feature importance results, where Network was the dominant group for streaming tuples prediction.

CPU utilization also distinguishes several clusters. Clusters 2, 3, 4, and 6 show very high CPU\_mean values, indicating processor-intensive profiles. Cluster 3 is the most resource-intensive overall, combining very high CPU utilization with the highest memory and power values. Input/output activity is most visible in cluster 1, while cluster 6 combines high CPU utilization with very low power values, suggesting a distinct low-power but compute-active profile.

Overall, Table 5 and Figure 81 show that streaming tuples executions form network-heavy, CPU-intensive, high-memory/high-power, input/output-intensive, and low-power profiles. These findings complement the feature importance analysis by showing that streaming throughput is mainly shaped by network behaviour and device capacity, with power, memory, and input/output features further distinguishing the workload profiles.

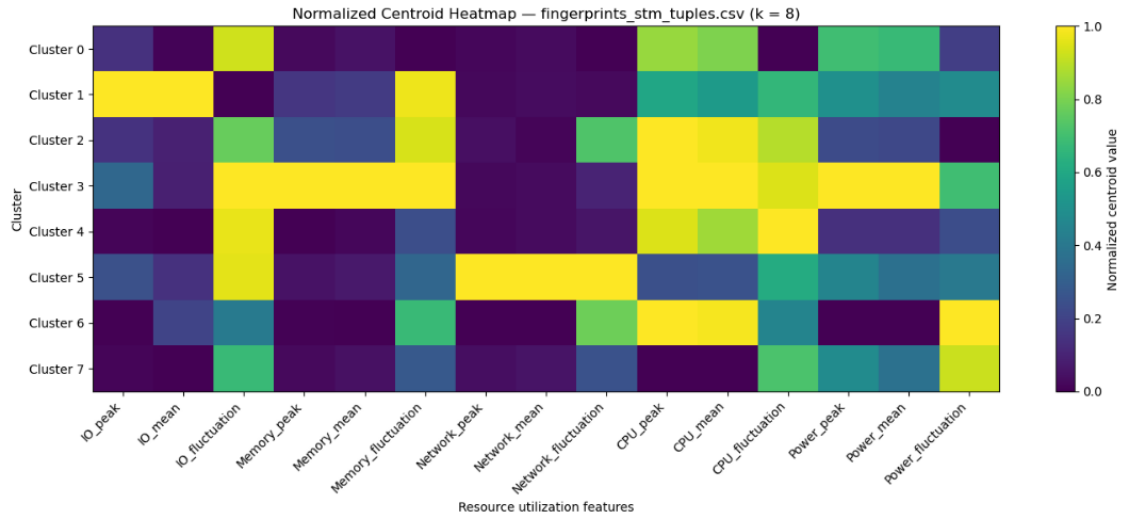


Figure 81. Normalized KMeans Centroid Heatmap for the Streaming Tuples Application Metric

| cluster | IO_peak       | IO_mean      | IO_fluctuation | Memory_peak  | Memory_mean  | Memory_fluctuation | Network_peak  | Network_mean  | Network_fluctuation | CPU_peak  | CPU_mean  | CPU_fluctuation | Power_peak | Power_mean | Power_fluctuation |
|---------|---------------|--------------|----------------|--------------|--------------|--------------------|---------------|---------------|---------------------|-----------|-----------|-----------------|------------|------------|-------------------|
| 0       | 45003.112065  | 992.153813   | 0.521739       | 3710.319513  | 3443.526683  | -0.956522          | 47865.703130  | 21277.811017  | -0.782609           | 90.911374 | 80.812196 | -0.304348       | 293.695652 | 264.162300 | -0.260870         |
| 1       | 151874.881185 | 19776.123507 | -0.777778      | 7734.831052  | 6673.306296  | 0.703704           | 52373.590459  | 24316.967885  | -0.740741           | 76.104615 | 63.135796 | 0.259259        | 213.851852 | 175.422604 | 0.037037          |
| 2       | 45257.244700  | 2519.111370  | 0.300000       | 10336.087485 | 8474.099785  | 0.650000           | 69425.771820  | 15088.416845  | 0.350000            | 99.914110 | 92.406940 | 0.450000        | 100.650000 | 87.345995  | -0.450000         |
| 3       | 68579.477625  | 2487.007938  | 0.625000       | 33147.108500 | 28559.314100 | 0.750000           | 49960.130700  | 22270.517713  | -0.625000           | 99.838600 | 93.834800 | 0.500000        | 421.750000 | 390.155537 | 0.250000          |
| 4       | 28633.461727  | 804.262673   | 0.575758       | 2911.542918  | 2625.006973  | -0.545455          | 49152.291579  | 20870.198388  | -0.696970           | 96.879942 | 84.331052 | 0.545455        | 62.945424  | 57.304115  | -0.212121         |
| 5       | 57877.698143  | 3532.439452  | 0.565217       | 4361.386617  | 3914.400952  | -0.391304          | 995682.203478 | 472319.384230 | 0.782609            | 55.728387 | 43.319404 | 0.217391        | 193.173913 | 146.915800 | -0.043478         |
| 6       | 27106.514784  | 4678.417628  | -0.200000      | 3041.526952  | 2115.239224  | 0.200000           | 30994.083084  | 8120.818000   | 0.440000            | 99.931544 | 93.000740 | 0.080000        | 5.087560   | 4.460776   | 0.560000          |
| 7       | 28766.712061  | 813.993239   | 0.173913       | 3713.147217  | 3294.200783  | -0.478261          | 68253.955391  | 32817.559774  | -0.391304           | 41.508752 | 26.354435 | 0.304348        | 206.028609 | 148.218065 | 0.478261          |

Table 5. Centroid Matrix for the Streaming Tuples Application Metric

#### 4.4.2.5 Summary of KMeans Clustering and Centroid Interpretation

Figure 82 presents the main findings from the KMeans clustering and centroid interpretation across the four application metrics. The results indicate that the fingerprint datasets exhibit distinct and interpretable resource-usage patterns rather than a single, uniform execution behaviour. Across all metrics, the most common recurring profiles are

CPU-intensive, input/output-intensive, network-heavy, and high-memory/high-power executions. The database metric is primarily distinguished by input/output activity and CPU demand, whereas the machine learning inference metric demonstrates the greatest centroid diversity, reflecting a wider range of resource-usage behaviours. The two streaming metrics, streaming latency and streaming tuples, display very similar clustering structures, which is expected since they originate from the same workload, despite capturing different aspects of performance. Overall, the clustering analysis complements the regression and feature-importance results by revealing clear workload behaviour patterns and illustrating how different applications place varying demands on system resources.

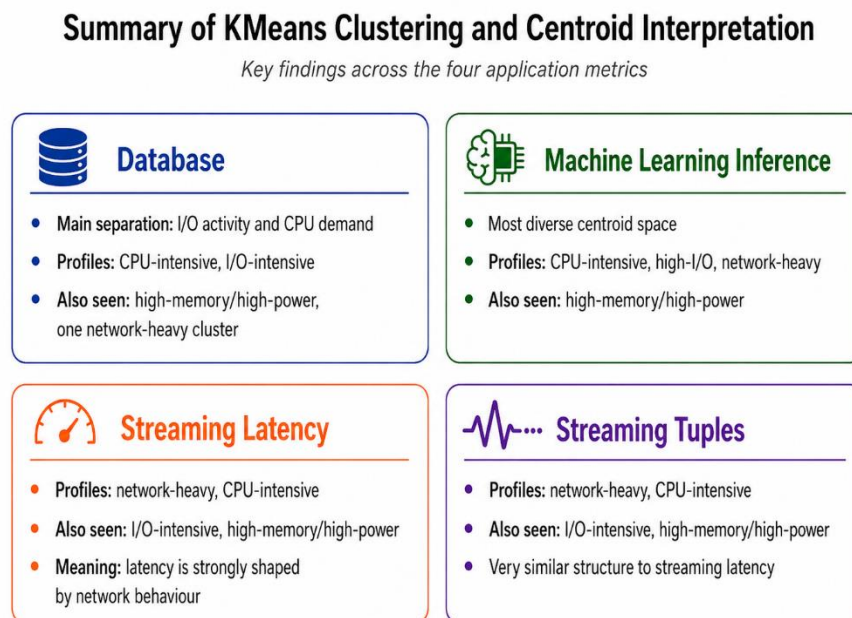


*Figure 82. Summary of KMeans Clustering and Centroid Interpretation  
Across Application Metrics*

## 4.5 BenchPilot Interface Evaluation

This section evaluates the BenchPilot interface by comparing the original BenchPilot experimentation workflow with the workflow supported by the developed interface. The comparison focuses on the main stages of experimentation, including experiment configuration, deployment, monitoring, result access, and post-experiment analysis. The goal is to assess how the interface reduces manual effort, improves usability, and supports more organized and reproducible experiment management.

### 4.5.1 Original BenchPilot Workflow

In the original BenchPilot workflow, deploying an experiment requires several manual steps before the experiment can be executed. First, the user must connect to the remote server through SSH using a terminal. After accessing the server, the user must identify the Docker container running BenchPilot by executing `docker ps`, and then enter the container using a command such as `docker exec -it <container_id> bash`. At the same time, the user may open the BenchPilot Jupyter environment in a browser through the exposed server IP address and port in order to manage files more easily.

Figure 83 shows the Jupyter file browser used in the original workflow. Although this environment allows the user to browse files, open notebooks, and manage experiment-related scripts, the deployment process still depends on terminal commands and manual file handling. As a result, users must switch between the terminal and the browser-based file interface when preparing, launching, and inspecting experiments.

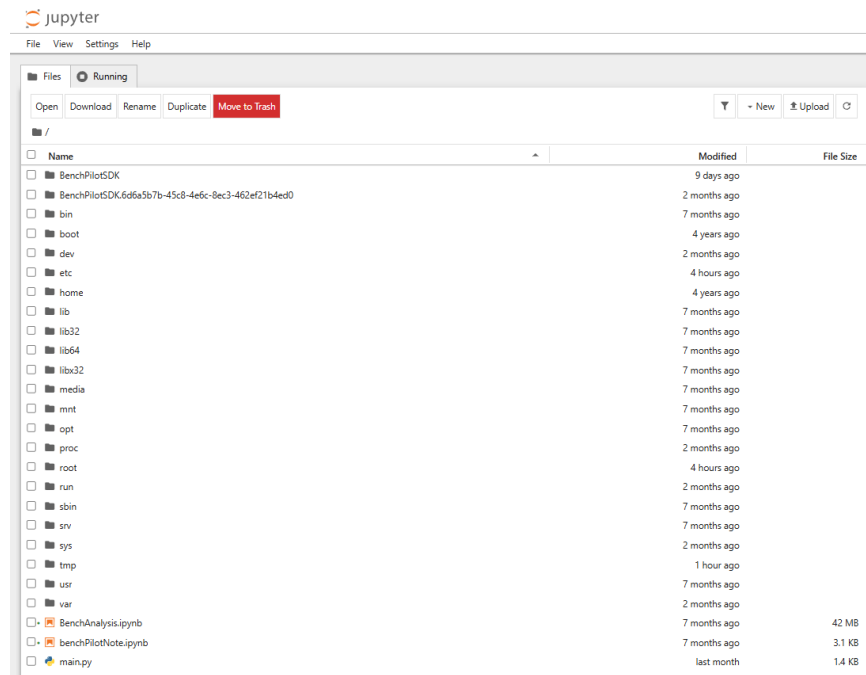


Figure 83. Jupyter file browser used in the original BenchPilot workflow

In the original BenchPilot workflow, cluster changes require direct modification of the cluster configuration file, located at `BenchPilotSDK/conf/BenchPilot-cluster.yaml`. This file defines the manager node and the available worker nodes, including their IP addresses, hostnames, authentication details, execution paths, hardware specifications, and suffix values. Therefore, users must understand the expected YAML structure and edit it carefully when adding or modifying nodes. A simplified example of this configuration is shown in Figure 84. This manual process increases the risk of formatting mistakes and inconsistent configuration, especially when multiple users share the same BenchPilot installation. In contrast, the developed interface allows cluster information to be viewed and managed through the dashboard, reducing the need to edit the cluster configuration file directly.

```

cluster:
  manager:
    ip: "<manager_ip>"
    hostname: "<manager_hostname>"
    proxies:
      http_proxy: "<proxy_url>"
      https_proxy: "<proxy_url>"
  nodes:
    - ip: "<node_ip>"
      hostname: "<node_hostname>"
      username: "<username>"
      password: "<password>"
      path: "<execution_path>"
      threads: "<number_of_threads>"
      cores: "<number_of_cores>"
      ram: "<ram_capacity>"
      suffix: "<node_suffix>"

```

Figure 84. Simplified BenchPilot cluster configuration file

In addition to editing the cluster configuration file, the original BenchPilot workflow also requires manual changes to the deployment script before running an experiment. As shown in Figure 85, the user must edit `main.py` to declare the experiment name in the `exp` list and update the paths for the metadata JSON file and the experiment YAML file. The user must also manually create the corresponding YAML experiment file in the declared path before executing the script. This adds another manual step to the workflow, since an incorrect experiment name or file path can prevent the experiment from running correctly or cause the outputs to be saved in the wrong location.

```

from BenchPilotSDK.controllers.controller import Controller

exp = [
    "mdb-stm-scpu_rpi5",
]

import sys
metadata_json_file = "/BenchPilotSDK/experiments/csyme02_experiments/final/rpi5/" + exp[int(sys.argv[1])] + ".json"
exp_file = "/BenchPilotSDK/conf/yaml_experiments/csyme02_experiments/final/rpi5/" + exp[int(sys.argv[1])] + ".yaml"

controller = Controller(bench_experiments_file=exp_file, debug_mode=True)
controller.start_experiment()
controller.export_benchmark_metadata(metadata_json_file, True)

print("*** Getting Metrics and Saving them into CSVs***")
from BenchPilotSDK.controllers.postExperimentController import PostExperimentController

postExpController = PostExperimentController()

postExpController.load_benchmark_meta(metadata_json_file)
experiments = postExpController.get_available_workloads_from_metadata()
exported_benchmarks = []
for experiment in experiments:
    exported_benchmarks.append({experiment: postExpController.get_benchmark_metrics(experiment, export_results_to_csv=True)})
else:
    print ("Passing this experiment for now")

```

Figure 85. `main.py`

Another manual step in the original BenchPilot workflow is the configuration of the CSV export path for post-experiment analysis. As shown in Figure 86, the user must edit `BenchPilotSDK/controllers/postExperimentController.py` and manually modify the `exported_result_path` variable to specify where the generated CSV file will be stored. This means that, before deployment, the user must define not only the experiment itself but also the output location for the post-experiment analysis results. As a result, incorrect path settings can lead to misplaced or missing CSV outputs and further increase the manual effort required by the original workflow.

```
107         if export_results_to_csv:
108             exported_result_path = "/BenchPilotSDK/experiments/csyeo02_experiments/final/xavier/csv/" if len(exported_result_path) <= 0 else
exported_result_path
109             df_exported_experiments.to_csv(exported_result_path + workload_name + ".csv")
```

*Figure 86. Manual CSV export path configuration in `postExperimentController.py`*

After completing the required manual configuration steps, the user can deploy the experiment from the terminal. From the directory containing `main.py`, the user launches the experiment as a background process using a command such as `nohup python3 main.py &`. This produces a `nohup.out` file, which contains the runtime logs of the experiment. To check whether the workloads have been deployed correctly as Docker services, the user must then run `docker service ls`. If more detailed information about a specific service is needed, the user must execute a command such as `docker service ps --no-trunc <service_id>`. Therefore, monitoring the deployment requires additional terminal commands and manual inspection of both log files and Docker service output.

After the experiment finishes, the user must manually locate the generated JSON files and inspect the workload logs to determine whether the experiment executed correctly. The generated CSV file must also be located and checked separately. In the original workflow, there are no automatic validation checks for workload outputs or CSV structure. If the user wants to analyze the CSV file in Jupyter, the CSV path must be copied manually and inserted into a notebook, which increases the possibility of using an incorrect or incomplete output file. In addition, if an experiment fails, some Docker

services may remain active because they are not always cleaned up automatically. In that case, the user must manually remove the remaining services using a command such as `docker service rm $(docker service ls -q)`. The developed interface handles this cleanup process automatically when stopping or managing experiments, reducing the manual effort required for recovery after failed deployments.

Overall, the original BenchPilot workflow requires users to coordinate several separate tools and files, including the terminal, Docker commands, YAML configuration files, Python scripts, Jupyter notebooks, JSON logs, and CSV outputs. While BenchPilot supports repeatable experiment execution, many surrounding tasks remain manual and fragmented. Users must manually configure cluster information, define experiment paths, launch deployments, inspect logs, check Docker services, validate outputs, and prepare CSV files for analysis. If an experiment fails, additional manual cleanup may also be required to remove remaining Docker services. These limitations motivate the need for the developed interface, which integrates configuration, deployment, monitoring, result access, cleanup, and post-experiment analysis into a single workflow.

As summarized in Figure 87, the original workflow involves multiple manual steps across different tools before, during, and after experiment execution.

**Summary of Manual Steps in the Original BenchPilot Workflow**

| Workflow stage                                                                                                         | Original BenchPilot workflow                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
|  <b>Server access</b>                 | Connect to the remote server through SSH                                                                                                     |
|  <b>BenchPilot environment access</b> | Identify the running Docker container using <code>docker ps</code> and enter it using <code>docker exec -it &lt;container_id&gt; bash</code> |
|  <b>File management</b>               | Use the Jupyter file browser separately from the terminal                                                                                    |
|  <b>Cluster configuration</b>         | Manually edit <code>BenchPilotSDK/conf/BenchPilot-cluster.yaml</code>                                                                        |
|  <b>Experiment declaration</b>        | Manually edit <code>main.py</code> and update the experiment list                                                                            |
|  <b>Output path configuration</b>     | Manually define metadata JSON, YAML, and CSV export paths                                                                                    |
|  <b>YAML experiment file</b>          | Manually create the experiment YAML file in the declared path                                                                                |
|  <b>Deployment</b>                    | Run the experiment from the terminal using <code>nohup python3 main.py &amp;</code>                                                          |
|  <b>Runtime logs</b>                  | Inspect the generated <code>nohup.out</code> file manually                                                                                   |
|  <b>Docker service monitoring</b>    | Run <code>docker service ls</code> and <code>docker service ps --no-trunc &lt;service_id&gt;</code>                                          |
|  <b>Failed experiment cleanup</b>   | Manually remove remaining services using <code>docker service rm \$(docker service ls -q)</code>                                             |
|  <b>Result inspection</b>           | Locate and inspect JSON logs and CSV outputs manually                                                                                        |
|  <b>Output validation</b>           | Manually check workload logs and CSV structure                                                                                               |
|  <b>Jupyter analysis</b>            | Copy the CSV path manually into a Jupyter notebook                                                                                           |

*Figure 87. Summary of manual steps in the original BenchPilot workflow*

#### 4.5.2 BenchPilot Interface Workflow

The developed BenchPilot interface replaces the manual and fragmented steps of the original workflow with a unified web-based process. Instead of connecting to the server through SSH, entering the BenchPilot container, editing configuration files, and launching experiments from the terminal, the user can perform the main experimentation tasks directly through the interface.

The workflow begins with user authentication. After logging in, the user is redirected to the dashboard, where the available cluster nodes, workload configuration section, and

navigation options are presented. From the dashboard, the user can inspect the current cluster state and then move to the Deploy Experiments page to create a new deployment.

To configure an experiment, the user first selects the target node from the list of available nodes. The experiment can then be added either through manual workload selection or through batch input mode, where multiple experiment names can be entered at once. After the configuration is saved, the selected experiments are added to the deployment queue. Before deployment starts, the interface displays a review window that allows the user to inspect and adjust the workload templates for the queued experiments. These edits are applied only to the selected deployment and do not modify the user's stored workload templates.

The interface also improves deployment safety. Before an experiment is added or launched, the system checks whether the selected node is already being used by another running experiment. If the node is busy, the interface displays a warning message and prevents the conflicting deployment. This reduces the risk of overlapping experiments on the same edge device, which could otherwise affect the validity of the collected results.

Once deployment begins, the user can monitor the experiment through the View Experiments page. This page shows experiment batches, execution status, running experiments, and Docker services from a single interface. Instead of manually inspecting `nohup.out` files or running Docker commands such as `docker service ls`, the user can view logs, Docker service information, and experiment progress directly from the browser. Running experiments can also be stopped from the interface, and the system handles the related cleanup process.

After an experiment completes, the interface provides direct access to the generated artifacts. The user can inspect the JSON log, CSV output, execution logs, error information, and generated YAML file through separate tabs. The interface also performs basic validation checks on workload outputs and CSV structure, helping users identify incomplete or suspicious results before using them for analysis. In addition, completed experiments can be reviewed manually by marking them as successful or failed, and users can add comments to record observations.

For post-experiment analysis, the interface integrates with Jupyter. When a completed experiment has a CSV output, the user can open it directly in a generated Jupyter notebook. The notebook already contains the CSV path and loading code, allowing the user to continue analysis without manually locating the file or writing the initial setup code.

Overall, the BenchPilot interface workflow reduces the need for manual file editing, terminal commands, Docker inspection commands, manual output validation, and manual notebook setup. It integrates experiment configuration, deployment, monitoring, result inspection, cleanup, and post-experiment analysis into a single workflow, making the experimentation process easier to manage and more suitable for repeated and multi-user edge benchmarking.

#### **4.5.3 Workflow Comparison**

The comparison between the original BenchPilot workflow and the developed interface shows that the interface reduces the number of manual actions required before, during, and after experiment execution. In the original workflow, users must coordinate several tools and files, including the terminal, Docker commands, configuration files, Python scripts, Jupyter notebooks, JSON logs, and CSV outputs. In contrast, the developed interface brings these operations into a single web-based workflow, as summarized in Figure 88.

### Comparison Between the Original BenchPilot Workflow and the Developed Interface Workflow

| Workflow stage                                                                                                     | Original BenchPilot workflow                                                                                         | BenchPilot interface workflow                                                 |
|--------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|
|  <b>User access</b>               | Connect to the server through SSH and enter the BenchPilot Docker container manually                                 | Log in through the web interface                                              |
|  <b>Cluster configuration</b>     | Manually edit <code>BenchPilotSDK/conf/BenchPilot-cluster.yaml</code>                                                | View and manage cluster information through the dashboard                     |
|  <b>Experiment configuration</b>  | Manually edit <code>main.py</code> and create the experiment YAML file                                               | Select node and workloads through the Deploy Experiments page                 |
|  <b>YAML generation</b>           | YAML file must be created manually in the declared path                                                              | YAML file is generated automatically by the backend                           |
|  <b>Output path configuration</b> | Metadata, YAML, and CSV paths must be edited manually                                                                | Output paths are created automatically inside the user workspace              |
|  <b>Deployment</b>                | Run the experiment from the terminal using <code>nohup python3 main.py &amp;</code>                                  | Deploy experiments using the interface                                        |
|  <b>Batch execution</b>           | Requires repeated manual deployment steps                                                                            | Multiple experiments can be added to a queue and deployed as a batch          |
|  <b>Runtime monitoring</b>        | Inspect <code>nohup.out</code> and run Docker commands manually                                                      | View logs, running experiments, and Docker services directly in the interface |
|  <b>Docker service inspection</b> | Use commands such as <code>docker service ls</code> and <code>docker service ps --no-trunc &lt;service_id&gt;</code> | Inspect Docker services and task details through the interface                |
|  <b>Conflict detection</b>        | User must manually check whether a node is already in use                                                            | Interface warns the user if the selected node is already busy                 |
|  <b>Failed experiment cleanup</b> | Remaining Docker services may need to be removed manually                                                            | Interface handles cleanup when stopping or managing experiments               |
|  <b>Result access</b>             | Locate JSON logs, CSV files, YAML files, and execution logs manually                                                 | Access JSON, CSV, YAML, logs, and errors from the experiment view             |
|  <b>Output validation</b>        | User manually checks workload logs and CSV structure                                                                 | Interface provides validation messages and warnings                           |
|  <b>Jupyter analysis</b>        | User manually copies the CSV path into a notebook                                                                    | Interface opens a generated notebook with the CSV already loaded              |
|  <b>Multi-user organization</b> | Shared files and manual tracking                                                                                     | User-specific workspaces and metadata records                                 |

Figure 88. Comparison between the original BenchPilot workflow and the developed interface workflow

The original workflow depends heavily on manual file editing, command-line execution, and direct inspection of output files. This increases the risk of configuration mistakes, incorrect paths, conflicting deployments, incomplete cleanup after failed experiments, and errors during result validation. The developed interface addresses these issues by allowing users to configure experiments, deploy individual or batch runs, monitor execution, inspect Docker services, access outputs, validate results, and open CSV files in Jupyter from the same environment.

#### 4.5.4 Summary

This section evaluated the developed BenchPilot interface by comparing it with the original BenchPilot workflow. The original workflow requires users to interact with several separate tools, including the terminal, Docker, YAML configuration files, Python scripts, Jupyter notebooks, JSON logs, and CSV outputs. As a result, experiment

configuration, deployment, monitoring, result validation, cleanup, and post-experiment analysis involve multiple manual steps.

The developed interface addresses these limitations by integrating the main experimentation tasks into a single web-based workflow. Users can manage cluster information, configure experiments, generate YAML files, deploy individual or batch experiments, monitor running executions, inspect Docker services, access generated outputs, validate results, and open CSV files in Jupyter directly from the interface. This reduces the need for manual file editing, terminal commands, and direct navigation through output directories.

Overall, the comparison shows that the BenchPilot interface improves usability, reduces manual effort, and supports more organized experiment management. These improvements are especially important in multi-user edge experimentation environments, where clear ownership, deployment safety, result traceability, and reproducibility are necessary for reliable benchmarking.

# Chapter 5: Conclusion

## 5.1 Summary of the Thesis

This thesis focused on the problem of understanding workload performance in multi-tenant edge computing environments. In such environments, several workloads may run on the same edge device and compete for limited resources. This makes performance harder to predict, since the behaviour of an application depends not only on its own workload characteristics, but also on the hardware platform, resource usage, and the other workloads running alongside it.

To study this problem, the thesis developed a workload fingerprint extraction pipeline. The purpose of this pipeline was to transform raw BenchPilot experiment outputs into structured records that could be used for analysis. Each fingerprint combines information about the active workloads, the device characteristics, the observed performance value, and summarized resource utilization metrics. The generated fingerprints were then separated into application-specific datasets for database performance, machine learning inference, streaming latency, and streaming tuples.

The thesis then used these datasets for performance modelling and workload analysis. Several regression models were trained and evaluated for each application metric. In addition, feature importance analysis was applied to examine which input features and resource categories influenced the predictions. Clustering was also used to group experiments with similar resource usage patterns, providing another way to characterize workload behaviour.

Alongside the data analysis part, the thesis also addressed the practical limitations of the original BenchPilot experimentation workflow. Although BenchPilot supports repeatable benchmarking, several tasks still require manual effort, including configuration editing, path management, terminal-based deployment, Docker inspection, output retrieval, and

notebook preparation. These steps become difficult to manage when many experiments, devices, and workload combinations are involved.

To improve this workflow, a web-based BenchPilot interface was designed and implemented. The interface extends the existing BenchPilot framework by providing a unified environment for experiment configuration, deployment, monitoring, result access, validation, and Jupyter-based analysis. Overall, the thesis combines a performance analysis pipeline with a practical system extension, supporting more organized and reproducible experimentation for co-located workloads in edge environments.

## 5.2 Main Findings

The first main finding is that workload fingerprints provide a useful representation for performance prediction in multi-tenant edge environments. The regression results showed that the extracted features can capture meaningful relationships between workload composition, hardware characteristics, resource utilization, and application-level performance. Tree-based and ensemble-based models performed better than Linear Regression, which suggests that workload performance is influenced by non-linear interactions rather than simple linear effects.

The prediction accuracy differed across application metrics. Database performance and machine learning inference were predicted with high accuracy, with Extra Trees achieving the strongest results for both. Streaming latency was more difficult to model, but Decision Tree still captured a large part of the performance variation. Streaming tuples was the most challenging target, although XGBoost and Extra Trees still produced useful predictions. This shows that fingerprint-based modelling is effective, but the difficulty of prediction depends on the type of workload metric being analyzed.

The feature importance results showed that different workloads depend on different resource dimensions. Database performance was mainly influenced by input/output activity and CPU-related features. Machine learning inference depended mostly on workload identity, power consumption, and CPU utilization. For the streaming metrics, network behaviour was the dominant factor, with CPU capacity and power also

contributing to streaming throughput. These findings show that workload fingerprints can help explain which resources matter most for each application type.

The clustering results showed that the experiments do not follow one uniform resource usage pattern. Instead, the datasets contain distinct workload profiles, such as CPU-intensive, input/output-intensive, network-heavy, high-memory, and high-power executions. This means that clustering can complement prediction by helping identify groups of experiments with similar runtime behaviour.

The evaluation of the BenchPilot interface showed that the developed system improves the experimentation workflow compared with the original process. The original workflow requires users to move between the terminal, Docker commands, configuration files, Jupyter notebooks, JSON logs, and CSV outputs. The interface brings these tasks into a single environment, reducing manual file editing, command-line interaction, direct output navigation, and manual notebook setup.

Overall, the findings show that this thesis contributes in two connected ways. The workload fingerprint pipeline supports performance prediction and workload characterization, while the BenchPilot interface makes the experimentation process easier to manage and more reproducible. Together, these results support the use of automated experimentation and fingerprint-driven analysis for studying co-located workloads in edge computing environments.

### **5.3 Limitations**

Although this thesis achieved its main objectives, several limitations should be acknowledged. These limitations mainly concern the scope of the collected dataset, the generality of the prediction models, the depth of the interface evaluation, and the current level of automation provided by the BenchPilot interface.

First, the fingerprint based analysis was performed using the experiments collected during this thesis. The dataset includes different workload categories, devices, and co-location scenarios, but it is still limited to the specific experimental environment and available

BenchPilot workloads. As a result, the findings may not fully generalize to all edge infrastructures, hardware platforms, workload types, or deployment conditions. A larger dataset with more devices, repetitions, workload combinations, and runtime conditions would provide stronger evidence about the general applicability of the models.

Second, the regression models were trained separately for each application metric, namely database performance, machine learning inference, streaming latency, and streaming tuples. This approach was necessary because each metric represents a different type of performance target. However, it also means that the trained models are specialized for specific workload categories. The thesis did not develop a single unified model that can predict all types of performance metrics across all workload types.

Third, although feature importance and clustering provided useful insight into workload behaviour, the analysis remains exploratory. The results identify important resource dimensions and workload profiles, but they do not yet translate directly into automatic scheduling or placement decisions. For example, the system can show that network behaviour is important for streaming workloads, but it does not yet use this knowledge to recommend where a workload should be deployed or which workloads should not be co-located.

The BenchPilot interface evaluation was also limited. The evaluation was based on a workflow comparison between the original BenchPilot process and the developed interface. This comparison showed that the interface reduces manual effort and integrates several fragmented tasks into a single environment. However, the evaluation did not include a formal user study with participants. Therefore, the results demonstrate functional and workflow improvements, but they do not measure user satisfaction, task completion time, or error reduction from the perspective of different users.

Another limitation concerns the validation mechanisms provided by the interface. The interface includes useful checks for workload outputs and CSV structure, but these checks are still basic. They help identify obvious missing outputs, invalid logs, or incomplete CSV files, but they do not replace deeper manual analysis of experiment quality. More

advanced validation would be required to automatically detect subtle performance anomalies or incorrect experimental conditions.

Finally, the interface improves experiment management but does not yet provide advanced decision making. It can prevent conflicting deployments on the same node, support batch execution, organize outputs, and simplify result access. However, it does not yet use the prediction models or clustering results to guide experiment planning automatically. Future versions could connect the analysis pipeline with the interface so that users receive performance predictions, interference warnings, or placement suggestions before launching experiments.

## **5.4 Future Work**

Several directions can build on the work presented in this thesis. One important direction is to expand the workload fingerprint dataset. Although the current datasets include different workload categories, devices, and co-location scenarios, a larger dataset would make the performance models more reliable and general. Future experiments could include more edge devices, additional workloads, more repetitions, different workload parameters, and more diverse co-location cases.

Another direction is to improve the modelling process. In this thesis, the models were trained separately for each application metric because each metric represents a different type of performance target. Future work could investigate whether more general models can be developed across multiple workload categories. It would also be useful to explore models that can be updated over time as new experiments are executed and new fingerprints are collected.

The prediction models could also be connected directly to the BenchPilot interface. For example, before deploying an experiment, the interface could estimate the expected performance of the selected workload combination. It could also warn users when a specific co-location scenario is likely to cause strong interference. This would allow the system to support experiment planning before execution, rather than only analyzing results after the experiment has finished.

The clustering and feature importance results could also be used to support workload placement decisions. Since the analysis identified workload profiles such as CPU-intensive, input/output-intensive, network-heavy, and high-power executions, future work could use these profiles to recommend which workloads are suitable to run together and which combinations should be avoided.

The BenchPilot interface can also be extended with more analysis and visualization features. Future versions could include experiment comparison pages, resource usage graphs, performance trend summaries, and automatic report generation. These additions would help users understand results more easily without relying on separate scripts or manual analysis.

The validation mechanisms of the interface could also be improved. The current checks help identify missing outputs, invalid workload logs, and incomplete CSV files. Future work could add more advanced checks for unusual performance values, abnormal resource usage, incomplete workload execution, or inconsistencies between JSON logs and CSV outputs.

Finally, a formal usability evaluation could be conducted with real users. The evaluation in this thesis compared the original BenchPilot workflow with the developed interface, but it did not include a user study. Future work could ask users to complete the same experiment management tasks with both workflows and compare task completion time, errors, and user feedback. This would provide stronger evidence about the practical usability benefits of the interface.

## 5.5 Final Remarks

This thesis showed that workload fingerprints can support the analysis and prediction of performance behaviour in multi-tenant edge environments. By combining workload identity, hardware characteristics, resource utilization, and application-level performance, the fingerprint representation provides a useful basis for both predictive modelling and workload characterization.

At the same time, the thesis showed that improving the experimentation workflow is essential for scalable and reproducible edge benchmarking. The developed BenchPilot interface reduces manual effort by integrating experiment configuration, deployment, monitoring, result access, validation, and Jupyter-based analysis into a single web-based environment.

Together, the fingerprint analysis pipeline and the BenchPilot interface provide a foundation for more efficient experimentation and future performance-aware workload management in edge computing environments.

## References

- [1] J. Georgiou, M. Symeonides, M. Kasioulis, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "BenchPilot: Repeatable & Reproducible Benchmarking for Edge Micro-DCs," in *Proc. 2022 IEEE Symp. Comput. Commun. (ISCC)*, Rhodes, Greece, 2022, pp. 1–6, doi: 10.1109/ISCC55528.2022.9912882.
- [2] J. Georgiou, M. Symeonides, G. Pallis, and M. D. Dikaiakos, "Automating Multi-Tenancy Performance Evaluation on Edge Compute Nodes," in *Proc. IEEE Int. Conf. Edge Comput. Commun. (EDGE)*, 2025, pp. 103–114.
- [3] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.
- [4] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, 2020, pp. 42–54.
- [5] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for Internet of Things realization," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2961–2991, 2018.
- [6] S. Wijethilaka and M. Liyanage, "Survey on network slicing for Internet of Things realization in 5G networks," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 2, pp. 957–994, 2021.
- [7] P. Sharma, L. Chaufourrier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study," in *Proc. 17th Int. Middleware Conf.*, Trento, Italy, 2016, pp. 1–13, doi: 10.1145/2988336.2988337.

- [8] N. Wang, M. Matthaiou, D. S. Nikolopoulos, and B. Varghese, "DYVERSE: DYnamic VERTical scaling in multi-tenant edge environments," *Future Gener. Comput. Syst.*, vol. 108, pp. 598–612, 2020.
- [9] R. Morabito, "Virtualization on Internet of Things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017, doi: 10.1109/ACCESS.2017.2704444.
- [10] G. Premsankar and B. Ghaddar, "Energy-efficient service placement for latency-sensitive applications in edge computing," *IEEE Internet Things J.*, vol. 9, no. 18, pp. 17926–17937, 2022.
- [11] I. Alla, H. B. Olou, V. Loscri, and M. Levorato, "From sound to sight: Audio-visual fusion and deep learning for drone detection," in *Proc. 17th ACM Conf. Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2024, pp. 123–133, doi: 10.1145/3643833.3656133.
- [12] K. Krasovitskiy, S. Christou, and D. Zeinalipour-Yazti, "LLM-MS: A multi-model LLM search engine," in *Proc. 2025 IEEE 41st Int. Conf. Data Engineering Workshops (ICDEW)*, Hong Kong, China, 2025, pp. 107–114, doi: 10.1109/ICDEW67478.2025.00020.
- [13] W. Chen, K. Ye, and C.-Z. Xu, "Co-locating online workload and offline workload in the cloud: An interference analysis," in *Proc. 2019 IEEE 21st Int. Conf. High Performance Computing and Communications; IEEE 17th Int. Conf. Smart City; IEEE 5th Int. Conf. Data Science and Systems (HPCC/SmartCity/DSS)*, Zhangjiajie, China, 2019, pp. 2278–2283, doi: 10.1109/HPCC/SmartCity/DSS.2019.00316.
- [14] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of Docker containers for interfering microservices," in *Proc. 2018 IEEE Int. Conf. Services Computing (SCC)*, 2018, pp. 33–40.

- [15] UCY LInC Lab, "BenchPilot: Getting Started." [Online]. Available: <https://ucy-linc-lab.github.io/BenchPilot/docs/getting-started/>. [Accessed: May 2026].
- [16] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.
- [17] S. Weisberg, *Applied Linear Regression*, 4th ed. Hoboken, NJ, USA: Wiley, 2014.
- [18] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA, USA: Wadsworth, 1984.
- [19] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
- [20] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, 2006.
- [21] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 2001.
- [22] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
- [23] H. Drucker, "Improving regressors using boosting techniques," in *Proc. 14th Int. Conf. Mach. Learn. (ICML)*, 1997, pp. 107–115.
- [24] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, San Francisco, CA, USA, 2016, pp. 785–794.
- [25] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, vol. 1, 1967, pp. 281–297.

- [26] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022.
- [27] R. Schwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Inf. Fusion*, vol. 81, pp. 84–90, 2022.
- [28] D. Chicco, M. J. Warrens, and G. Jurman, "The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation," *PeerJ Comput. Sci.*, vol. 7, p. e623, 2021.
- [29] C. J. Willmott and K. Matsuura, "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance," *Clim. Res.*, vol. 30, no. 1, pp. 79–82, 2005.
- [30] A. Fisher, C. Rudin, and F. Dominici, "All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously," *J. Mach. Learn. Res.*, vol. 20, no. 177, pp. 1–81, 2019.
- [31] R. L. Thorndike, "Who belongs in the family?" *Psychometrika*, vol. 18, no. 4, pp. 267–276, 1953.
- [32] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, 1987.
- [33] Meta Open Source, "Your First Component," React Documentation. [Online]. Available: <https://react.dev/learn/your-first-component>. [Accessed: May 9, 2026].
- [34] S. Ramírez, "FastAPI," FastAPI Documentation. [Online]. Available: <https://fastapi.tiangolo.com/>. [Accessed: May 9, 2026].

**[35]** MongoDB, Inc., "Documents," MongoDB Manual. [Online]. Available: <https://www.mongodb.com/docs/manual/core/document/>. [Accessed: May 9, 2026].

**[36]** MongoDB, Inc., "Data Modeling in MongoDB," MongoDB Manual. [Online]. Available: <https://www.mongodb.com/docs/manual/data-modeling/>. [Accessed: May 9, 2026].