UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

METADATA CAPTURE, KNOWLEDGE EXTRACTION, AND PREDICTIVE ANALYSIS FOR HPC WORKFLOWS

NICOLÒ GIACOMINI

Thesis supervisor
RAÜL SIRVENT PARDELL (Barcelona Supercomputer Center (BSC))

Tutor: ROSA MARIA BADIA SALA (Department of Computer Architecture)

Degree
Master's Degree in Innovation and Research in Informatics (Computer Networks and Distributed Systems)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Master's Thesis

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona (FIB)
Master in Innovation and Research in Informatics
Computer Network and Distributed Systems

Master's thesis

Metadata Capture, Knowledge Extraction, and Predictive Analysis for HPC Workflows

Nicolò Giacomini

1. Supervisor Raül Sirvent Pardell
Workflows and Distributed Computing
Barcelona Supercomputing Center

2. Tutor Rosa María Badia Sala
Department of Computing Architecture
Universitat Politècnica de Catalunya

Nicolò Giacomini
Metadata Capture, Knowledge Extraction, and Predictive Analysis for HPC Workflows
Master's thesis, January 27th, 2026
Supervisors: Raül Sirvent Pardell and Rosa María Badia Sala

Master's Thesis
Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona (FIB)
Master in Innovation and Research in Informatics
Computer Network and Distributed Systems

Abstract

In High-Performance Computing (HPC) environments, the difficulty to accurately predict workflow execution times often leads to inefficient resource allocation. To resolve this challenge, this thesis shows a structured pipeline implemented in the COMPSs framework. The system captures metadata to perform predictive analysis, which provides more reliable execution time estimation for complex computational tasks.

The research presents three primary contributions. First, a robust, non-intrusive profiler was developed to capture performance metrics across heterogeneous environments. By implementing an adaptive fallback mechanism and standardizing output into the RO-Crate format, the profiler ensures data interoperability with negligible runtime overhead. Second, the architecture of the Provenance Storage system was rethought to include a REST API middleware and Docker containerization. These structure updates simplify remote connectivity and integration with external services.

Finally, the research evaluates standard models, such as linear regression, polynomial regression and random forests. Afterwards, the research advances to more complex architectures like Feedforward Neural Networks and XGBoost. These advanced models incorporate specialized preprocessing and theoretical refinements. The study develops a hybrid framework that integrates the Universal Scalability Law (USL) with machine learning. This physics-informed approach corrects theoretical deviations, demonstrating higher accuracy and resilience than purely data-driven models in HPC environment.

Acknowledgement

This thesis has been possible due to the work of the Workflows and Distributed Computing Group of the Barcelona Supercomputing Center. I work in a incredible team full of smart people who helped me to grow as a professional and as a person. Especially, I want to thank Raül Sirvent Pardell, my supervisor who assist me during the development of this project and Rosa Maria Badia Sala, my tutor for giving me the chance to work on this project. This experience has allowed me to learn new aspects of the field, from the technical details of the COMPSs framework to the collaborative workflow in scientific research. I am thankful for the opportunity to have conducted my research in an environment that encourages both innovation and continuous learning.

Contents

1 Introduction

In recent years, research in the field of machine learning has developed rapidly, enabling researchers to design models capable of learning from increasingly large and complex datasets. This progress has been achieved during the consequent adoption of artificial intelligence (AI) systems in different sectors. According to recent global data, approximately 40% of companies worldwide are actively utilizing AI in their operations, while another 42% are exploring its potential integration [14]. In total, over 82% of businesses are engaging with AI in some form in 2025, indicating the pervasive influence of intelligent systems on organizational practices.

This accelerated adoption reflects the broad applicability of AI in solving real-world challenges, such as optimizing decision-making, improving predictive accuracy, automating repetitive processes, and enabling data-driven innovation. The diffusion of AI technologies extends across domains including healthcare, manufacturing, education, and logistics, where machine learning algorithms are employed to detect patterns, forecast outcomes, and support complex planning tasks. Moreover, national adoption rates vary considerably: for example, 59% of companies in India have already embraced AI technologies, compared to 33% in the United States [14]. These figures highlight the global but uneven nature of AI integration.

Despite this widespread deployment, the scientific community continues to face critical challenges regarding interpretability, bias, and generalization. Many models remain opaque ("black boxes"), making their decision processes difficult to validate or reproduce [18]. Addressing these challenges is essential to ensure responsible, transparent, and reliable AI systems. Consequently, research now emphasizes explainability, robustness, and ethical alignment, aiming to bridge the gap between empirical performance and theoretical understanding.

In parallel, the growing complexity of scientific workflows, particularly in high-performance and data-intensive computing environments, has brought renewed attention to the issues of provenance and reproducibility. As experiments become increasingly automated, distributed, and reliant on heterogeneous computational resources, the ability to capture, represent, and reuse the knowledge generated throughout the research lifecycle has become essential for modern scientific practice [19, 9]. Provenance information—describing the origin, transformation, and

dependencies of data and processes—plays a crucial role in ensuring transparency, reproducibility, and long-term reusability of scientific results.

Motivated by these observations, the objective of this thesis is to *develop and evaluate methodologies for the automated capture, representation, and analysis of provenance and experimental metadata in scientific workflows*. This work seeks to contribute to the field by integrating provenance modeling, knowledge extraction, and predictive analytics techniques to support reproducible research and facilitate the interpretation of computational experiments. In particular, the thesis explores how structured metadata and machine learning can be combined to yield actionable insights from experimental traces. As a result, this improves both the reproducibility and the understanding of complex computational processes.

The motivation for this study arises from the growing interest and need in the fields of research, data analysis, knowledge extraction, and machine learning. Through this Master's program, I had the opportunity to achieve a solid foundation in these topics, which was further reinforced by experience at the Barcelona Supercomputing Center (BSC).

In many current systems, data storage is constrained by limitations such as a lack of a standardized format and poor interoperability across different platforms. These inherent drawbacks complicate reproducible research, primarily due to the difficulty in tracing data provenance, which subsequently inhibits the ability to reproduce experiments.

The primary objective of this Master's thesis is to develop a profiling system that provides researchers with a common and interoperable format for the archival and analysis of experimental results. The stored data is aggregated into standardized containers, referred to as Crates, which facilitates sharing and ensures ease of use and reproducibility across other systems.

The project is structured into three main stages. The first stage involves implementing the profiling system, which captures the main resource information during application execution. This mechanism is essential for providing a comprehensive overview of the executed application and the resource consumption associated with a specific experiment. The second stage focuses on data storage and the deployment of a database. To this end, a dedicated database has been deployed to make the stored data accessible to the wider research community, specifically the BSC community. The final stage is the development of a predictive machine learning tool. This stage is dedicated to forecasting key performance metrics based on historical execution data from the system. Specifically, the initial focus is on predicting resource usage

and execution time for a specific application, given its input data and configuration parameters.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the related works and the tools and standards adopted in the project. Chapter 3 details the implementation of the metadata profiling during execution, shows the database architecture, and includes an analysis of these profiling tools. Chapter 4 presents the machine learning models and the experimental results and their interpretation, and Chapter 5 concludes with a summary of findings and future research directions.

1.1 Related Work

The challenge of predicting application runtimes and resource usage in computing environments has been a long-standing area of research. This section reviews previous work in job runtime prediction, focusing on the use of historical data, job similarity, and the application of machine learning techniques.

Machine Learning and Feature-Based Approaches

In the recent study conducted in [22], they aim to develop a system that predicts the execution time of high-performance computing (HPC) jobs. This is a challenging task due to the variability in job characteristics and user estimations. One critical aspect the authors focus on is how job runtimes are represented and inputted into machine learning models for prediction. Traditionally, users provide an estimated runtime, but these estimates are often inaccurate, therefore the prediction performance are poor and inefficient. To address this drawback, the paper proposes a novel approach where job runtimes are categorized into distinct classes, and users select a runtime class rather than specifying an exact runtime. These runtime classes must be well-defined to accurately capture the distribution and patterns of job runtimes in the specific HPC environment. To identify the correct expected time, they implemented a genetic algorithm (GA) to automatically search through the possible runtime class definitions and identify the set of classes that yield the best prediction performance. Along this algorithm, they implemented different machine learning (ML) models, including K-Nearest Neighbors (KNN), Support Vector Regression (SVR), Deep Neural Networks (DNN), and Extreme Gradient Boosting (XGB). In this approach, the GA treats each set of runtime class boundaries as a 'chromosome'. The quality of a chromosome is measured by the performance of an ML model using those boundaries. The GA then evolves its population of chromosomes over iterations, applying natural selection, crossover, and mutation to find progressively better class definitions. In summary, the genetic algorithm is used to solve the difficult and critical subproblem of defining

runtime classes for better machine learning input, which in turn enhances the overall goal of providing an accurate, practical HPC job execution time prediction system.

The foundational concept of using historical data to forecast future behavior was demonstrated in [8]. In their study of Unix systems, they showed that past resource usage data is a valuable predictor. They also highlighted that the similarity between different applications is a key factor for a precise estimation. The problem of defining this similarity is not trivial, as jobs must be correctly categorized to enable accurate predictions. In [24], the word "template" is central in this context and refers to a configured set of key job characteristics that are used to categorize and identify similar jobs. These templates specify which attributes are considered relevant to partition the job history into categories where applications are evaluated to be similar. By defining templates that capture the main parameters influencing runtime, jobs can be grouped effectively into meaningful clusters. The discussed paper extends earlier work by employing search techniques (greedy and genetic algorithms) to discover effective templates automatically, optimizing the selection of job characteristics used to define similarity. Their experiments demonstrate that these data-driven template selection methods improve prediction accuracy significantly compared to fixed or manually chosen templates.

In another study [25], Tanash et al. studied how to predict job resources (memory and time) for submitted jobs on HPC environments. They utilized large datasets from two different university systems totaling millions of job instances. The historical data provided by the Slurm workload manager have served to implement supervised machine learning models from Scikit-Learn library [21] and the neural network with PyTorch [20]. In the paper, they specifically noted the importance of the account, which represents the user or group submitting the job. They found that different accounts often have distinct usage patterns and developed a greedy algorithm to find optimal combinations of these accounts for model training.

Provenance and RO-Crate standard

As regarding the standard of the metadata and their structure, the work described in [15] presents an important advancement in the representation and packaging of provenance information for computational workflow executions. The followed model is RO-Crate, an evolving standard for bundling research data with metadata. Their contribution, termed Workflow Run RO-Crate (WRROC), innovatively extends the base RO-Crate [7] framework and Schema.org vocabulary to capture both prospective and retrospective provenance of workflows in a machine-actionable manner. This facilitates traceability, reproducibility, and interoperability across diverse workflow management systems (WMSs), addressing critical challenges arising from heterogeneity in provenance representations and execution environments. The

WRROC model is composed of three progressively detailed profiles (Process Run Crate, Workflow Run Crate, and Provenance Run Crate) which cater to varying levels of provenance granularity without enforcing rigid implementation demands. This flexible profiling supports simple command executions, orchestrated workflows, and detailed step-level execution metadata, respectively. The practical impact of this work is underpinned by multiple community-driven implementations across different WMSs (including COMPSs [26]) and an associated Python toolkit (runcrate) that supports conversion, inspection, and re-execution of RO-Crates. These implementations embody the interoperability promise of WRROC by enabling provenance comparison and workflow reproduction across heterogeneous systems and workflow languages, as demonstrated in digital pathology and machine learning use cases.

An important research that most of the work on this master thesis, has based on is [23]. Early efforts in provenance for workflows distinguished between prospective provenance (workflow definitions) and retrospective provenance (execution details), typically capturing basic execution metadata such as user, time, and environment. Attempts to leverage semantic web technologies like RDF and OWL aimed to improve interoperability but introduced complexity for non-expert users. Here, our work contributes to increase the amount of metadata of the retrospective provenance in order to describe a detailed picture of every execution and provide many useful data for both users and machines. The development of standards solidified provenance representation for interoperability. In the research, they show that the metadata has to follow the FAIR guiding principles [23] (Findable, Accessible, Interoperable, and Reusable), which also extend to computational workflows. Thanks to the JSON-LD format, they describe Research Object Crate (RO-Crate) as the ideal standard for packaging research data and metadata, gaining adoption across diverse scientific domains. Finally, in [23] they also demonstrate the efficient implementation in COMPSs programming model, where this method automatically records provenance without user annotations, minimizing runtime overhead in supercomputing environments.

2 Background

2.1 COMPSs Framework

This research study was conducted in collaboration with the Workflows and Distributed Computing team at the Barcelona Supercomputing Center (BSC). The work specifically aligns with the objectives of the Provenance, Metadata, and Reproducibility research group. A significant component of this collaboration involved leveraging the group's central project: COMP Superscalar (COMPSs) [16].

COMPSs is a high-performance framework and workflow management system developed by the Workflows and Distributed Computing group. This established technology directly supports the objectives of the present research due to its inherent flexibility and advanced features for the distributed execution of different kinds of applications. This capability is particularly valuable within High-Performance Computing (HPC) environments, such as the BSC. Furthermore, the widespread adoption of COMPSs in numerous teams within BSC and the utilization of this framework by many other research centers validate its robustness and ease of integration, making it an ideal foundation for the new features developed in this thesis.

Given this context, COMPSs presented a strategically sound choice for the present research. The development of the framework within the affiliated research group provided direct access to the source code, allowed a practical extension and customization to meet the specific requirements of this study. Furthermore, the support of the team during the development guaranteed optimal integration and troubleshooting.

COMPSs features broad language support, natively supports Java, and provides bindings for Python (PyCOMPSs [26]), C/C++, and R, which significantly contributes to its diffused adoptability across diverse research teams. Furthermore, a key architectural strength of COMPSs is its platform independence, achieved by clearly separating the application logic from the underlying distributed infrastructure. This separation ensures that applications are highly portable and can be readily deployed across heterogeneous computing environments.

The framework consists of two principal components: the programming model and the runtime system. Both components are briefly described in the following

sub-sections. For the scope of this work, the focus is on the Python implementation of COMPSs, known as PyCOMPSs. The thesis involves extending its provenance capturing functionality through targeted modifications to the source code of both the runtime system and the Python language binding to fully support comprehensive provenance recording.

2.1.1 Programming model

The COMPSs task-based programming model is designed to abstract the complexities of parallel execution from the developer. This is achieved by allowing applications to be written in a conventional, sequential programming style. To leverage parallelization within COMPSs, the programmer has only to annotate standard Python methods with the framework's decorators.

This approach ensures that the parallelization and distribution overhead (including processes such as thread creation, data distribution, and message passing) is automatically managed by the runtime system. The definition of a parallel task in Python is executed by applying the `@task` decorator immediately preceding the method definition. Both functions and methods (instance or class) can be selected for remote execution as tasks. When defining these tasks, the programmer must supply the runtime with explicit hints regarding the parameter handling and return value properties, which guide the framework's resource management and scheduling decisions.

Although primitive data types (like integers, floats, booleans, and strings) are automatically recognized, files and special Python objects should be explicitly declared in the task decorator. Some examples for non-primitive types supported by COMPSs are: `FILE`, `DIRECTORY`, `COLLECTION`, `COLLECTION_FILE`, `DICTIONARY`, `STREAM`.

The default direction of parameters is read-only (IN); however, it is also possible to declare them as write (`OUT`), read-write (`INOUT`), concurrent (`CONCURRENT`), or commutative (`COMMUTATIVE`)[1].

If the method returns one or more values, the number (or type) of return values should be provided through the returns keyword added to the task decorator, for example, `returns=2` or `returns=(int, list)`.

Listing 2.1 and 2.2 show two very simple examples of task definitions in Python. Listing 2.1 takes one primitive type as input and calculates the square of that input.

[1]Read the documentation to know more in detail `https://compss-doc.readthedocs.io`

Listing 2.2 is an accumulator that takes a collection and calculates the sum of every two elements until the whole list is reduced to one final value.

```
1     @task(returns=int)
2     def square(x):
3         return x * x
4
```

Listing 2.1: Definition of `square`

```
1     @reduction(chunk_size="2")
2     @task(returns=int, values=
      COLLECTION_IN)
3     def sum_reduce(values):
4         total = 0
5         for val in values:
6             total += val
7         return total
8
```

Listing 2.2: Definition of `sum_reduce`

When the square method is invoked within the main loop (as shown in Listing 2.3), it can be executed in parallel because the square calculation for each list element is independent of all others. The concurrency potential is visually represented in the dependency graph generated post-execution (Figure 2.1). While this graph illustrates the application's theoretical parallelization capacity, the actual concurrent execution is ultimately constrained by the availability of computational resources. Following the computation of the square of each list element, the reduce method is responsible for aggregating the partial results by calculating the sum in a pairwise manner.

```
1  def main():
2      squared_values = []
3      # Launch 10 independent tasks, one for each i = 0..9
4      for i in range(10):
5          result = square(i)
6          squared_values.append(result)
7          sum_of_squares = sum_reduce(squared_values)
8          sum_of_squares = compss_wait_on(sum_of_squares)
9          print("Sum of squares:", sum_of_squares)
10
```

Listing 2.3: PyCOMPSs program to calculate the sum of squares

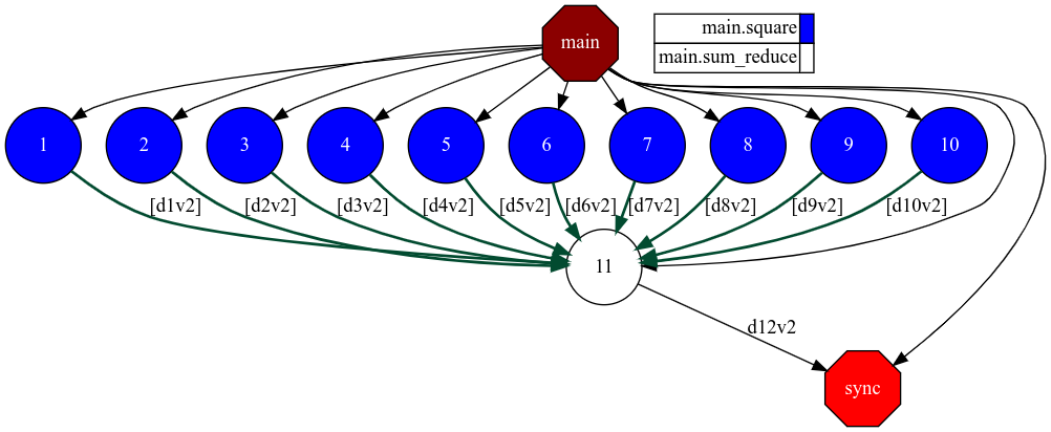


Fig. 2.1.: Dependency graph of the squares application

While this simple example illustrates the fundamental parallelization capabilities of COMPSs, the full spectrum of the framework's features only becomes evident within the context of complex applications[2].

The tracking of COMPSs is strictly limited to the sections of the application that are marked with the `@task` decorator. The information provided within the decorator and the function parameters is managed effectively by the framework, moving the data execution from the user code to a remote node. However, the runtime system lacks visibility regarding the parameters or execution of standard Python methods.

2.1.2 Runtime system

The COMPSs runtime system, implemented in Java, performs several crucial functions. It is responsible for the automatic discovery of data dependencies between annotated methods, the dynamic construction of the task dependency graph, and the subsequent scheduling of each task based on data readiness and resource availability.

After the launch of a COMPSs-enabled sequential application, every call to an annotated method is intercepted and forwarded to the runtime system before its execution. This process results in the creation of new asynchronous tasks. If necessary, the data required for execution is fetched from the remote location where the task was initially created.

The task dependency graph, as visually represented in Figure 2.1, is constructed on the fly. It can be exported in SVG format by enabling the graph generation

[2] `https://compss-doc.readthedocs.io/en/stable/Sections/07_Sample_Applications.html`

flag `-graph` during application execution. It is noteworthy that graph generation is automatically activated when the provenance generation feature is enabled.

Given the extensive and intricate codebase of COMPSs, a substantial portion of the work conducted for this thesis involved in-depth study and navigation of the source code. This comprehensive understanding was essential for adapting the framework's functionalities to meet the specific requirements of the present research.

2.2 Execution Environment and Available Tools

This section illustrates the computational environment and tools utilized for this project. While the primary development and testing were conducted on the **MareNostrum 5 (MN5)** supercomputer, it was also essential to prove that the project could port and run correctly on different machine architectures. This environment is hosted at the **Barcelona Supercomputing Center (BSC-CNS)**, which provides a heterogeneous collection of world-class HPC resources.

2.2.1 Barcelona Supercomputing Center

The Barcelona Supercomputing Center stands as one of the most renowned high-performance computing (HPC) research institutions in the world. The BSC is a consortium managed by the Spanish Ministry of Science, the Catalan Government, and the Universitat Politècnica de Catalunya (UPC). The research within the institution is structured into four primary departments:

- **Computer Sciences**: Focuses on developing the supercomputing technology of tomorrow, from programming models and performance tools to new processor architectures.
- **Life Sciences**: Applies HPC to genomics, proteomics, and personalized medicine, including drug discovery and complex biological system simulations.
- **Earth Sciences**: Utilizes supercomputing for climate modeling, weather prediction, and understanding geophysical phenomena.
- **Computer Applications in Science and Engineering (CASE)**: Develops and optimizes complex simulation codes for fields ranging from fluid dynamics and materials science to social sciences.

In the center, we can find the "MareNostrum" supercomputers, which have made Spain and Europe major players in global supercomputing. This legacy continues with their newest system, MareNostrum 5, a powerful machine that is highly flexible for many different objectives in the research fields.

2.2.2 MareNostrum 5

Inaugurated in late 2023, MareNostrum 5 (MN5) is a flagship EuroHPC supercomputer that represents a monumental improvement in computational power for European researchers. With a total peak performance of 314 Petaflops (314×10^{15} floating-point operations per second), it consistently ranks among the top 15 most powerful supercomputers in the world.

However, the defining technical characteristic of MareNostrum 5 is not just its raw power, but its heterogeneity. Unlike "monolithic" systems designed for a single type of task, MN5 is an aggregation of specialized and interconnected partitions. This design acknowledges that modern science has different computational needs. A problem in classical physics simulation, a deep-learning AI model, and a data-intensive genomics analysis all require different types of computational resources. MN5 is built to serve all of them efficiently.

2.2.3 Partitions in MareNostrum 5

MareNostrum 5 is a heterogeneous system-of-systems, meaning it's built from several different specialized parts (partitions) that work together:

- **The General Purpose Partition (GPP):** This is a massive partition (over 717,000 CPU cores) built by Lenovo. It uses powerful Intel Xeon CPUs and is designed for traditional HPC simulations, such as climate modeling or complex engineering.

- **The Accelerated Partition (ACC):** This is the most powerful part of the system, built by Eviden. It gets its speed (approx. 230 Petaflops) from thousands of NVIDIA H100 GPUs. It is specialized for AI, deep learning, and other GPU-heavy calculations.

- **Next Generation Technology (NGT) Partitions:** These are two smaller, experimental systems. They serve as a testing partition for researchers to develop software for future hardware, including one partition with NVIDIA (ARM) CPUs and another with new Intel GPUs.

- **Interconnect and Storage:** A very high-speed NVIDIA InfiniBand network connects all the partitions. The data is managed in a multi-level system that includes 248 Petabytes of Active Storage for running jobs and 400 Petabytes of Archive Storage for long-term backups.

- **Green Computing (Efficiency):** The entire system is designed to be highly energy-efficient. It uses a Direct Liquid Cooling system (warm water) instead of traditional air conditioning. The heat captured from the processors is then reused to heat nearby university buildings.

2.2.4 Second-Stage Validation on Complementary BSC Platforms

The main development and testing of the project have been conducted on the MareNostrum 5 (MN5) system. However, a second stage of validation was performed on different systems. The goal of this phase was to ensure the project's portability, robustness, and performance consistency across different hardware architectures and production environments hosted at BSC, too.

The following complementary computational platforms were used for this purpose:

Nord4 This is a small part of the old MareNostrum 4. This partition is a general-purpose cluster based on Intel Xeon processors. It is a legacy part and now it is used as an alternative platform where to execute tests in research.

Nord3v2 Derived from the former MareNostrum 3 system, this partition utilizes an older generation of Intel Xeon architecture.

CTE-AMD The **C**ompute **T**est **E**nvironment (CTE) is equipped with AMD processors. This platform was instrumental in assessing the performance and portability of the developed solutions across different CPU microarchitectures.

Thanks to these available systems, it was possible to guarantee that the project was portable and interoperable with different architectures.

2.3 RO-Crate

RO-Crate [7] is a community-driven specification designed for the packaging and serialization of research artifacts and their associated metadata (e.g., datasets,

workflows, and publications) into a standardized format. The fundamental purpose of this specification is to enable research outputs to respect the FAIR principles (Findable, Accessible, Interoperable, and Reusable).

Moreover, RO-Crate leverages Schema.org vocabulary for its definitions. Schema.org[3] is a standard for structured data on teh web that RO-Crate utilizes. The key word standardized in the vocabulary are introduced in the JSON-LD (JavaScript Object Notation for Linked Data) metadata. JSON-LD extends the standard JSON format to incorporate Linked Data principles, so that it facilitates the robust representation of relationships between entities identified using IRIs (Internationalized Resource Identifiers). This choice of metadata format is critical, as it enhances the semantic interoperability of the packaged research objects.

2.3.1 JSON-LD motivation

The **JSON-LD** (JavaScript Object Notation for Linked Data) format is the core metadata serialization format for RO-Crate, since it provides simplicity, semantic rigor, and compliance with modern web standards. This choice yields four key strategic advantages:

Common and accessible syntax

JSON-LD maintains the common JSON syntax, which is one of the most widely adopted data interchange formats today. This format ensures that the metadata is easily accessible and parsable by both automated software agents and human researchers.

Semantic interoperability

A principal advantage of JSON-LD is its inherent ability to map keys to established vocabularies, such as *Schema.org*. This mechanism enables researchers to achieve true semantic interoperability by defining the metadata element meanings and relationships with standardized terms. Thanks to these common schemas, the system ensures that research metadata is consistently understood and processed across different tools.

Linked Data compliance and integration

JSON-LD is a designated Linked Data compliant format. This means that JSON-LD documents can be easily parsed into Resource Description Framework (RDF) triples, which are the foundational data model of the Semantic Web.

[3] `https://schema.org/`

Portability and Packaging

The format supports relative identifiers and contextual definitions, and it is suited for data packaging. These features allow the metadata to be logically and physically contained within the RO-Crate. This robust portability ensures that the metadata travels with the research artifacts and simplifies deployment across different computing environments.

2.3.2 Structure of RO-Crate Metadata

The core metadata for an RO-Crate is systematically stored within a single file named `ro-crate-metadata.json`. This file is required to be a valid JSON-LD document. Its foundational structure is governed by two essential key-value pairs:

- `@context`: this element provides a definitive reference to the JSON-LD context document (e.g., `https://w3id.org/ro/crate/1.2/context`). The context is critical since it defines the necessary term mappings, linking simple key names (e.g., `name`) to their corresponding fully-qualified Internationalized Resource Identifiers (IRIs) within established vocabularies like *Schema.org*.

- `@graph`: this is the central component, implemented as a JSON array. It contains the complete set of discrete **entities** described within the Crate. These entities include all primary research artifacts, including data files, computational workflows, associated software, human agents (people), and environment resource data.

Each individual entity object defined within the `@graph` array is characterized by the following required and descriptive properties:

- `@id`: a unique identifier assigned to the entity, typically expressed as an **absolute URI** or a **relative identifier** pointing to a file or directory within the Crate.

- `@type`: specifies the **class or type** of the entity (e.g., `Dataset`, `File`, `Person`), leveraging terms from recognized vocabularies (e.g., Schema.org) to establish semantic meaning.

- **Descriptive Properties**: these include standard metadata fields such as `name`, `description`, `author`, and `hasPart`, providing rich, human-readable information about the artifact.

Linked Data Model and Interoperability

Cross-references between these distinct entities are systematically expressed using nested objects that contain only the `@id` field of the referenced entity. This mechanism is central to the Linked Data philosophy of RO-Crate, offering several structural and operational benefits:

- **Flat and modular structure**: by utilizing these identifiers for referencing, the metadata structure avoids deep nesting. This results in a flat and modular structure where each entity is defined comprehensively only once, therefore the readability and the maintenance improve.

- **Graph representation**: the resulting model fundamentally represents a graph, where each entity acts as a node (e.g., dataset, file, or person), and the descriptive properties (or predicates) like `author` or `hasPart` represent the **edges**.

This graph-based model allows the tools to interpret and inspect the metadata using its **semantic structure**. This structure enhances discoverability and search efficiency within large volumes of research data, and at the same time, promotes semantic interoperability across different research systems.

2.3.3 RDF: Converting RO-Crate to Triples for Semantic Reasoning

A critical advantage of utilizing the JSON-LD format is the capability to transform the RO-Crate metadata into RDF (Resource Description Framework) triples. This transformation is not simply a format change; it is the key to interoperability. RDF is the standardized data model of the Semantic Web, which converts metadata contained in Crate into triples, allows it to be understood, queried, and, above all, integrated with any other RDF-compliant dataset, regardless of its origin or schema.

The RDF model expresses all the information as a set of discrete subject-predicate-object statements, or "triples." This structure may appear simple, but it is profoundly powerful. It provides the essential foundation for semantic reasoning, an automated process where new actions and relationships can be inferred from the metadata. For example, if an associated ontology defines that a `schema:File` is a subclass of a `schema:CreativeWork`, a semantic reasoner can automatically deduce that `data/file1.txt` is also a `schema:CreativeWork`, even if this fact was never explic-

itly stated in the original RO-Crate. This ability to discover and materialize implicit knowledge is the main advantage of linked data.

The following JSON-LD entity snippet 2.4 from an RO-Crate:

```
{
  "@id": "data/file1.txt",
  "@type": "File",
  "name": "Example Data File"
}
```

Listing 2.4: Example of JSON-LD in RO-Crate

is directly mapped into the equivalent RDF triples. The JSON-LD context ensures that terms like `File` and `name` are expanded to their full semantic definitions (e.g., from Schema.org), resulting in the following way 2.5:

```
<data/file1.txt> rdf:type schema:File .
<data/file1.txt> schema:name "Example Data File" .
```

Listing 2.5: Example of RDF triple

Once converted, this complete graph of triples can be loaded and managed in specialized graph databases known as triple stores (such as Apache Jena Fuseki, as discussed in Chapter 3.6). This dedicated storage mechanism unlocks the ability to query the whole metadata graph using the standard RDF query language, SPARQL (SPARQL Protocol and RDF Query Language).

This capability represents an important step beyond simple keyword searches. A keyword search could find a file by matching the string "Example Data File," but it cannot understand the context or relationships of that file. In contrast, SPARQL allows complex and graph-traversal queries supported by the semantic structure. An investigator can ask sophisticated questions like: "Find all Crates executed by authors from 'University X' using the input for the software '`analysis_tool.py`'." This ability to query across interconnected entities and inferred relationships is what allows users to discover data.

2.3.4 SPARQL: Querying the RO-Crate Graph

Once the RO-Crate metadata is converted into an RDF graph, SPARQL is the standard query language used to explore it. While it's used to "retrieve data," its real power lies in its ability to query the graph structure itself. Instead of searching for keywords,

SPARQL allows you to ask complex questions based on the relationships between entities.

A SPARQL query is built to find patterns in the RDF graph. A typical query consists of:

- **PREFIX declarations**: These are convenient shortcuts or aliases for the long URIs used as namespaces (like schema: for `https://schema.org/`).
- **A SELECT clause**: This specifies which variables (the "blanks" in your pattern, starting with ?) you want to see in the final results table.
- **A WHERE clause**: This is the core of the query. It defines the graph pattern you are searching for. This pattern is a set of triple templates, where variables are used to find matching data.

Example: Querying Actions in a Workflow Crate

This query searches for all "Actions" recorded in the crate and retrieves their associated instruments and start/end times.

```
PREFIX schema: <https://schema.org/>
SELECT ?action ?instrument ?start ?end
WHERE {
  ?action a schema:Action ;
          schema:instrument ?instrument ;
          schema:startTime ?start ;
          schema:endTime ?end .
}
```

Listing 2.6: Example of SPARQL query to get actions

How this query 2.6 works: The `WHERE` clause defines a pattern. It looks for any node (which we'll call ?action) that has an `rdf:type` (shortened to a in SPARQL) of `schema:Action`. The semicolons (;) are a shortcut to continue describing the same subject (?action). The query then finds the nodes connected to ?action via the schema:instrument, `schema:startTime`, and `schema:endTime` properties, storing their values in the corresponding variables. This single query effectively traverses multiple relationships for all matching actions.

Example: Identifying the RO-Crate Root

This more advanced query 2.7 demonstrates how to find the root Dataset entity of the RO-Crate itself by identifying the metadata file that describes it.

```
PREFIX schema: <http://schema.org/>
SELECT ?crate ?metadataFile
WHERE {
  ?crate a schema:Dataset .
  ?metadataFile schema:about ?crate .
  FILTER(CONTAINS(STR(?metadataFile), "ro-crate-metadata.json"))
}
```

Listing 2.7: Example of SPARQL query to identify the RO-Crate Root

This query consists of two parts. First, it finds all subjects that are a (a) `schema:Dataset`. Second, it finds another subject (identified by `?metadataFile`) that has a `schema:about` property pointing to our `?crate`. This finds the "reverse" relationship. Finally, the `FILTER` clause applies a conventional string search to ensure this `?metadataFile` is the specific one we're looking for. This query says, "Find all datasets that are the subject of an 'about' property from a file named 'ro-crate-metadata.json'."

3 Profiling and metadata capture of application

This chapter provides an analysis of the profiling tools chosen for this study. It will first detail the criteria and process behind the selection of the most suitable profiling methodologies. Following the selection process, the chapter will examine the performance characteristics of the selected tools. Next, the chapter focuses on the visual representation of the collected performance data, illustrating how the plots are generated and interpreted to meaningful metrics. Finally, the chapter concludes with a discussion of the data architecture, detailing how the profiling data are stored within the database, keeping the RO-Crate standards.

3.1 Evaluation of profiling tools

The profiler (`profiler.py`) is designed with a three-tiered methodology to ensure both robustness and accuracy across different environments. The active mode is selected based on the machine configuration (`profiler_config.json`).

- **Mode 1: `psutil` (main profiler)** This is the primary mode for general purpose machines. It uses the `psutil` Python library to gather system-wide metrics for CPU, memory, disk I/O, and network. Its cross platform nature and rich, high level API make it the preferred choice.

- **Mode 2: `top` (profiler fallback)** If `psutil` is not available, the system falls back to parsing the output of the `top` command. This provides a less detailed, but still functional, set of system-wide CPU and memory metrics.

- **Mode 3: `cgroup` (legacy system fallback)** This specialized mode is for shared/HPC environments (e.g., Slurm, Docker). It reads virtual files directly from the Linux Control Group (cgroup) filesystem. This is a critical distinction, as it measures the resource consumption of *only the specific job*, not the entire machine.

3.1.1 Comparison between the tools

The profiling system was implemented primarily using the `psutil` Python library. As a fallback mechanism, the system also integrates the `top` command to ensure functionality on machines where `psutil` is not available. For the HPC environment, besides these two approaches, another profiler which serves of `cgroup` kernel feature has been implemented to use a legacy functionality to profile some old HPC systems (e.g. Nord4). This multiple approach guarantees robustness and compatibility across a wide range of computing environments.

A common question is why a low-level tool like `perf` was not used. Even though `perf` works on the kernel level, while `psutil` works on the user level, for the project, `psutil` has been revealed as the perfect tool. While `perf` is extremely powerful and precise for its purpose, it was an unsuitable choice for our accounting script for the following practical reasons:

- **Purpose and Overhead:** `perf` is a low-level event sampler and using it for continuous, interval-based accounting is the wrong tool for the job. The file-based approach of `cgroup` (reading `cpuacct.usage`) or `psutil` is the kernel's intended, low-overhead mechanism for resource accounting.

- **Permissions and Security:** `perf` requires elevated privileges (or careful adjustment of `perf_event_paranoid`) to access kernel counters. This is often forbidden by default on production HPC clusters for security reasons. Our `cgroup` and `psutil` modes work with standard user privileges.

- **Complexity in HPC Environments:** As our text correctly notes, tracking the correct PIDs for a complex, multi-process workflow (like COMPSs) is difficult. Furthermore, to get job-specific data, `perf` would need to be made cgroup-aware. Our 'cgroup' mode, by definition, is already scoped to the job, making it a much simpler and more robust solution.

- **Portability:** For the system-wide modes, `psutil` is inherently cross-platform, whereas `perf` is tightly coupled to the Linux kernel.

- **Mismatch of Purpose and Metrics:** The script's goal is to record broad, high-level metrics (e.g., total CPU usage percentage, memory utilization, disk and network I/O bytes). `psutil` provides these metrics directly through a simple Python API. In contrast, `perf` is a diagnostic tool focused on low-level CPU events (e.g., instructions executed, cache misses, branch predictions). It does not provide the same range of high-level I/O and memory usage metrics.

- **Implementation Complexity:** Integrating `perf` would require managing an external `subprocess`, parsing the text output, and handling it as a non-interactive daemon. This is significantly more complex than the chosen methods:
  - **`psutil`:** A native Python library call.
  - **`top`:** A simple parsing of the command.
  - **`cgroup`:** A simple file-read operation (e.g., reading `cpuacct.usage`).
- **The PID and Job Scope Problem:** The profiler must be able to capture the total usage of a complex, multi-process job.
  - Using `perf` would require tracking all PIDs spawned by the job, which is impractical.
  - The `cgroup` mode solves this problem correctly. By reading the cgroup accounting files, the script measures the resource consumption of the *entire job* (the container), regardless of how many processes it spawns. This is the kernel intended mechanism for job-level accounting.

In conclusion, `psutil` supports a broad range of metrics, including CPU, memory, disk, and network usage, through a unified interface. This eliminates the need to maintain separate scripts specific for each platform for different operating systems. By combining portability, accuracy, and ease of integration, `psutil` offers a pragmatic and maintainable solution for system profiling in diverse execution environments.

3.2 Methodology for system resource profiling

The monitoring framework detailed in the scripts (`profiler.py`, `utils.py`) is a tool for tracking computer performance. Its primary function is to periodically sample and record system-level resource utilization (i.e. CPU, memory, network I/O, and disk I/O) to a standardized CSV file. The interval can be defined using an environment variable `COMPSS_PROFILING_INTERVAL`, which by default is set to 5 seconds. This value has been chosen because it is a compromise between overhead and completeness of the profiler results. However, this value can be adjusted by the user based on the kind of application and duration.

Its design allows it to be executed on heterogeneous systems by implementing three distinct data-collection methodologies, which are selected at runtime based on the configuration of the host machine.

3.2.1 General Operation and Configuration

The profiler is designed to run as a persistent background process, parallel to the main computational workload. The code is integrated in the COMPSs project here[1].

1. **Initialization:** Upon launch, the profiler first establishes signal handlers. It listens for termination signals (such as `SIGTERM` or `SIGINT`), which trigger a graceful shutdown. This ensures that the main profiling loop is cleanly exited and all file handles are properly closed.

2. **Configuration:** The behavior of the script is determined by the `profiler_config.json` file. At startup, the profiler identifies the host where the execution is running and consults this JSON file to select one of three profiling modes, called: `psutil`, `top`, or `cgroup`.

3. **Data Collection Loop:** The profiler enters a main loop that repeats at a configurable interval (e.g., 5 seconds). During each cycle, it gathers data using the selected mode, formats it into a CSV log file (`resource_profiling_{hostname}.csv`).

4. **Standardized Output:** Regardless of the collection method, the profiler standardizes its output to a 9-column CSV format for consistent data analysis: `CPU,MEM,BYTE_SENT,BYTE_RECV,BYTE_READ_DISK,BYTE_WRITE_DISK, TIME_READ_DISK,TIME_WRITE_DISK,TIME`

3.2.2 Profiling Modes

The key feature of the system is its ability to select the best profiling tool for the environment.

Mode 1: `psutil` library

This is the most comprehensive mode, used on systems where the `psutil` Python library is available. It provides a full, system-wide overview of all resources.

[1] `https://github.com/bsc-wdc/compss/blob/stable/compss/runtime/scripts/system/profiling/`

- **CPU:** Uses `psutil.cpu_percent()`. This function provides the average CPU utilization across all cores during the specified interval.

- **Memory:** Uses `psutil.virtual_memory().percent` to get the current system-wide memory usage.

- **Disk & Network I/O:** This mode excels at measuring I/O. It uses `psutil.disk_io_counters()` and `psutil.net_io_counters()`. These functions provide *cumulative* counters. The profiler records the values from the previous measurement and calculates the difference to determine the number of bytes read/written during that specific interval.

Mode 2: `top` command

This is a fallback mode used when `psutil` is unavailable. It relies on executing and parsing the output of the standard Linux `top` command.

- **CPU & Memory** It runs the command `top -b -n 1` and uses an `awk` script to parse the text output, extracting the system-wide CPU and Memory percentages.

- **Limitations** This method is less precise than `psutil` and, most importantly, **does not capture any disk or network I/O information**. The corresponding columns in the CSV file are filled with `None` or zeros.

Mode 3: `cgroup`

In this last fallback mode used in legacy systems, the profiler is able to detect the information about CPU, memory and disks. This method is more complex because it must calculate metrics that cgroups does not provide automatically. The next section details how this process works.

3.2.3 Implementation of `cgroup` profiler

This is the most specialized mode, and it solves a critical problem in modern computational environments, particularly in High-Performance Computing (HPC) or containerized systems.

Profiling on legacy infrastructure

Standard monitoring tools like `psutil` and `top` typically rely on specific system file formats to report resource usage. However, on the **nord4** machine, these tools proved ineffective, consistently reporting values of 0%. This issue arise from the

machine legacy architecture and older Linux kernel, which is not compatible with modern monitoring libraries.
The presence of **cgroup v1** (Control Group version 1) on Nord4 confirms the use of an older kernel generation. To overcome the failure of standard tools, we bypassed the high-level interfaces and utilized cgroups directly. This method allowed us to retrieve raw resource accounting data from the system

Control Group (cgroup)

Linux **cgroup** is a kernel feature that allows processes to be grouped and their resource usage to be isolated, limited, and **accounted for**. When a job is submitted to a cluster (e.g., via Slurm) or run in a container (e.g., Docker), it is placed into its own cgroup.
The kernel then exposes this cgroup specific resource usage as virtual files in the filesystem (usually under `/sys/fs/cgroup/`). The `cgroup` profiling mode is designed to read these specific virtual files to measure the resource consumption of **only the cgroup it is running in**.

Data Collection Process in `cgroup`

This mode does **not** measure system-wide usage. It measures the resource consumption of the job itself.

1. **Initialization:** The profiler first identifies the total resources of the *node* to serve as a baseline for percentage calculations.

   - It finds its cgroup file paths by parsing `/sys/fs/cgroup`.
   - It gets the total node memory ($\text{MemTotal}_{\text{KB}}$) by reading `/proc/meminfo`.
   - It gets the total node CPU cores ($\text{Cores}_{\text{node}}$).
   - It gets the multiplication factor calculating:

     $$\text{MutiplicationFactor} = \frac{\text{LogicalCores}}{\text{PhysicalCores}}$$

     This will be used to calculate the true value of CPU usage, since the hyperthreading is not used during the executions.

2. **Memory measurement (instantaneous):** At each interval, the profiler reads the file `/proc/meminfo`. This file contains the current, instantaneous memory

(in bytes) used by all processes in the cgroup. The percentage is then calculated against the total node memory:

$$\text{Mem}_{\%} = \left(\frac{\text{MemTotal}_{\text{KB}} - \text{MemAvailable}_{\text{KB}}}{\text{MemTotal}_{\text{KB}}} \right) \times 100$$

3. **CPU measurement (cumulative delta):** This is the most complex part. The profiler reads the file `cpuacct.usage` for cgroup v1, or `cpuacct.stat` in the case of cgroup v2. These files do **not** contain a percentage, but a cumulative, monotonically increasing counter of the total CPU time, in nanoseconds (ns), that the cgroup has consumed since its creation. The profiler reads the metric `usage_usec` that corresponds to the total CPU time consumed by all tasks in the cgroup. This is the sum of time spent in user space and time spent in kernel space (system space). To find the CPU usage within a specific interval, the profiler must take two measurements and calculate the delta, similar to the I/O calculation in `psutil`.
   For simplicity, let's take as an example the case of an old machine that has installed cgroup v1.

   Let:

   - T_1 = Wall-clock time of the first measurement.
   - T_2 = Wall-clock time of the second measurement (e.g., $T_1 + 5$ seconds).
   - C_1 = Value of `cpuacct.usage` (in ns) read at T_1.
   - C_2 = Value of `cpuacct.usage` (in ns) read at T_2.

   The script then performs the following calculation:

   Step 1: Find the elapsed interval time in nanoseconds between the init of the program and the next iteration.

   $$\Delta T_{\text{interval}} = (T_2 - T_1) \times 10^9$$

   Step 2: Find the CPU time consumed in nanoseconds.

   $$\Delta C_{\text{cpu}} = C_2 - C_1$$

Step 3: Calculate the average number of CPU cores used during the interval. The calculation comes from the division of the time spent in the CPUs during the elapsed interval. This value provides the average number of CPUs the job was using.

$$\text{Cores}_{\text{avg}} = \frac{\Delta C_{\text{cpu}}}{\Delta T_{\text{wall}}}$$

For example: If $\Delta T_{interval}$ is 5 seconds (5×10^9 ns) and the job consumed 10 seconds of CPU time ($\Delta C_{cpu} = 10 \times 10^9$ ns), the result for $Cores_{avg}$ would be 2.0. This means the job utilized, on average, 2 full CPU cores during that 5-second interval.

Step 4: Calculate the final percentage relative to the entire node. The final percentage is the number of cores the job used divided by the total cores available on the node.

$$\text{CPU}_{\%} = \frac{\text{Cores}_{\text{avg}}}{\text{Cores}_{\text{node}}} \times \text{MutiplicationFactor} \times 100$$

Example. Let's assume that in a machine there are 10 cores and with the hyper-threading the total number of cores is 12 ($Cores_{node} = 12$, considering the virtual cores), then the MultiplicationFactor $= \frac{12}{10} = 1.2$. Therefore, if $Cores_{avg} = 2$, the result is $CPU_{\%} = \frac{2}{12} \times 1.2 \times 100 = 20\%$.

Mode Limitations of `cgroup`

This mode provides invaluable job-specific data but does not capture disk or network I/O. These metrics are managed by different cgroup controllers (`blkio` and `net_cls/net_prio`) which this script does not query. Therefore, in this mode, the disk and network columns of the CSV are filled with zeros.

3.2.4 Summary of Modes

Mode	Data Source	Scope	CPU	Mem	Net I/O	Disk I/O
`psutil`	`psutil` library	System-Wide	✓	✓	✓	✓
`top`	`top` command	System-Wide	✓	✓	✗	✗
`cgroup`	`cgroup` tool	Job-Specific	✓	✓	✗	✗

Tab. 3.1.: Summary of profiling modes and their capabilities.

This fallback design ensures that the profiler can run successfully on an HPC cluster, on a legacy server, and on a laptop. These three alternatives (`psutil`, `top`, and `cgroup`) provide the most relevant data available in each environment.

3.3 Standard and Format of the Recorded Data

The Resource Usage Profiling is a tool used to get additional information about the status of the resources of a machine during the execution of an application. There are different reasons why this is important and why it required this an important update:

- Limited amount of data: before, the amount of data recorded during the execution was very low, only some information about the methods used, and some information about the tasks, but not information about the machine. This would be very useful for understanding how the machine behaves during the execution of a specific application.

- Lack of standard format: before, the small amount of data recorded was stored in a JSON format, without a standard. This is very limiting since the data cannot be used in an interoperable environment and the structure is not standardized, making it complex to cooperate with other tools and people (especially in research).

On the other side, for the provenance, we are already using RO-Crate standard, so our idea was to exploit the standard to store more data. This would guarantee to store the metadata in a standardized way and combine it with the already present data, giving a complete picture of the workflow.

3.3.1 The goals

In this first step, the goals are the following:

- Translate the statistical information to RO-Crate format: in the beginning, the data is stored in a simple JSON file; the goal is to move that data to a RO-Crate format [7].

- Add new statistical data: besides the data about the execution time of the methods, the idea is to extend the performance metrics, including data about CPU load, memory usage, and I/O operations.

- Centralize the statistical data in a database: this step is important for guaranteeing that the data is stored and can be used for analysis, training of the models, and other useful operations.

3.3.2 Baseline technology

The technologies used for achieving the goals are the following

- Provenance[23]: the chronology of the origin, development, ownership, location, and changes to a system or system component and associated data. Need to record metadata of a Workflow Provenance (data + software). It represents the history of a digital object, in particular it shows and tracks where data originated, how it was developed, and who owns it. In the context of computational workflows, this concept extends beyond merely saving the final results; it requires recording both the input and output data alongside the specific software and tools used to process it. By capturing this complete picture, we ensure that experiments remain reproducible and that future users can understand exactly how a calculation was derived.

- RO-Crate[7]: RO-Crate (Research Object Crate) is a standardized format for packaging research data with its metadata in a structured way. It generates a "crate" that includes the data files and metadata about the context. This format guarantees:

  - Standardized format that allows interoperability

  - Metadata respects the FAIR principles (Findable, Accessible, Interoperable, Reusable)

- Workflow Hub: Workflow Hub is a platform for sharing, discovering, and reusing computational workflows, especially in the life sciences. It allows researchers to publish and access workflows, providing a repository of documented and reusable processes.

- psutil library: a useful Python library used to get information about the resources of the machine. This will be useful for getting the information during the execution of an application.

In order to get the data, as explained in the previous section, the profiler is executed by the COMPSs framework when the “provenance” flag is enabled. In the end of the execution, the profiler generates some log files which track the evolution of the resources during the execution. In the log folder, the user can find a folder called “stats” which contains the CSV files containing all the samples of the resources status during the execution. Each CSV file in this folder corresponds to a specific

computation node, as the profiler runs on every node where the application is executed.

The data stored are the following:

- CPU usage: indicates the percentage of processing power being utilized by the application across nodes.
- Memory usage: shows the amount of RAM consumed during execution, helping identify potential memory bottlenecks.
- Disk usage: shows the amount of data read from and written to disk, useful for detecting I/O-intensive operations.
- Network usage: measures the volume of data transmitted and received, which is crucial for distributed applications.

After the collection of the data, the provenance runtime generates some plots that show the resource status evolution during the experiment. Moreover, if more nodes are used, the plots of CPU and memory usage are combined, showing the evolution of the respective resources among the nodes.

This is an example of CSV file generated by the profiling tool:

```
CPU,MEM,BYTE_SENT,BYTE_RECV,BYTE_READ_DISK,BYTE_WRITE_DISK,
TIME_READ_DISK,TIME_WRITE_DISK,TIME
0.2,15.0,46360,35526,0,20480,0,0,2026-01-09 11:00:43
0.2,15.0,25824,27581,0,16384,0,0,2026-01-09 11:00:48
0.2,15.0,20688,26496,0,4096,0,1,2026-01-09 11:00:53
0.2,15.0,24298,31451,0,20480,0,0,2026-01-09 11:00:58
0.2,15.0,95183,98705,0,20480,0,0,2026-01-09 11:01:03
0.2,15.0,318053,322674,0,20480,0,0,2026-01-09 11:01:08
0.2,15.0,32282,38744,0,16384,0,0,2026-01-09 11:01:13
0.2,15.0,38690,43424,0,4096,0,0,2026-01-09 11:01:18
5.0,15.8,2869504,4889920,0,45056,0,1,2026-01-09 11:01:23
94.0,17.5,3593340,980947,2826240,1003520,12,0,2026-01-09 11:01:28
100,17.5,23196,28816,151552,127741952,29,899,2026-01-09 11:01:33
100,17.5,279244,82911,0,9789440,0,14,2026-01-09 11:01:38
100,17.5,76262,39312,0,36864,0,0,2026-01-09 11:01:43
100,17.5,25296,30077,0,28672,0,0,2026-01-09 11:01:48
100,17.5,141158,141280,0,16384,0,1,2026-01-09 11:01:53
100,17.5,141872,145078,0,0,0,0,2026-01-09 11:01:58
```

```
100,17.5,362079,372395,0,26329088,0,100,2026-01-09 11:02:03
100,18.5,358871,829754,12288,16297984,0,23,2026-01-09 11:02:08
```

Listing 3.1: Example of CSV file obtained by the profiler

3.3.3 New standard used

Before this integration, the standards used were the following:

- Process Run Crate (Describes execution of tools in a computation): used to describe the execution of one or more tools of the same computation
- Workflow Run Crate (Captures workflow-driven execution of tools): similar to Process Run Crate, but assumes that the execution of the tools is driven by a computational workflow

After the new updates, this is the new RO-Crate satisfied standard:

- Provenance Run Crate (Adds details on internal workflow steps): extends Workflow Run Crate with guidelines for describing the internal details of each step of the workflow. This is achieved, thanks to:
  - New section called *resourceUsage* containing all the new metrics
  - Added vocabulary containing the terms of the new metrics in RO-Terms

The Key benefits of Provenance Run Crate are:

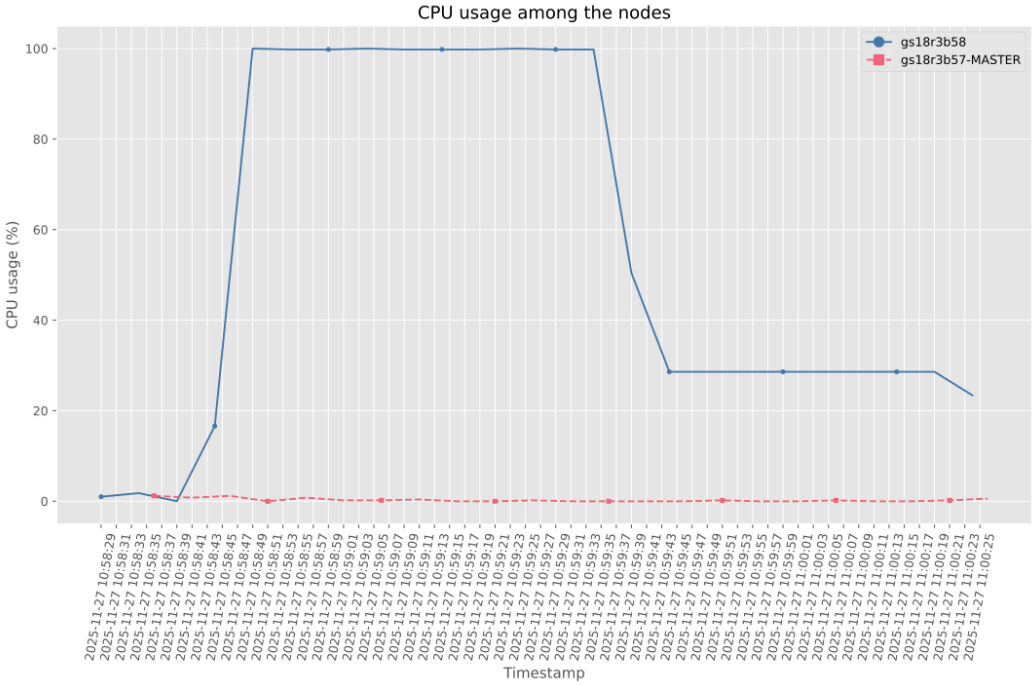
- Detailed provenance: captures internal steps of workflows
- Improved reproducibility: allows a deeper understanding of computational results
- Greater transparency: increases trust in data processing

In this appendix A, there is a snippet code example of *ro-crate-metadata.json*.

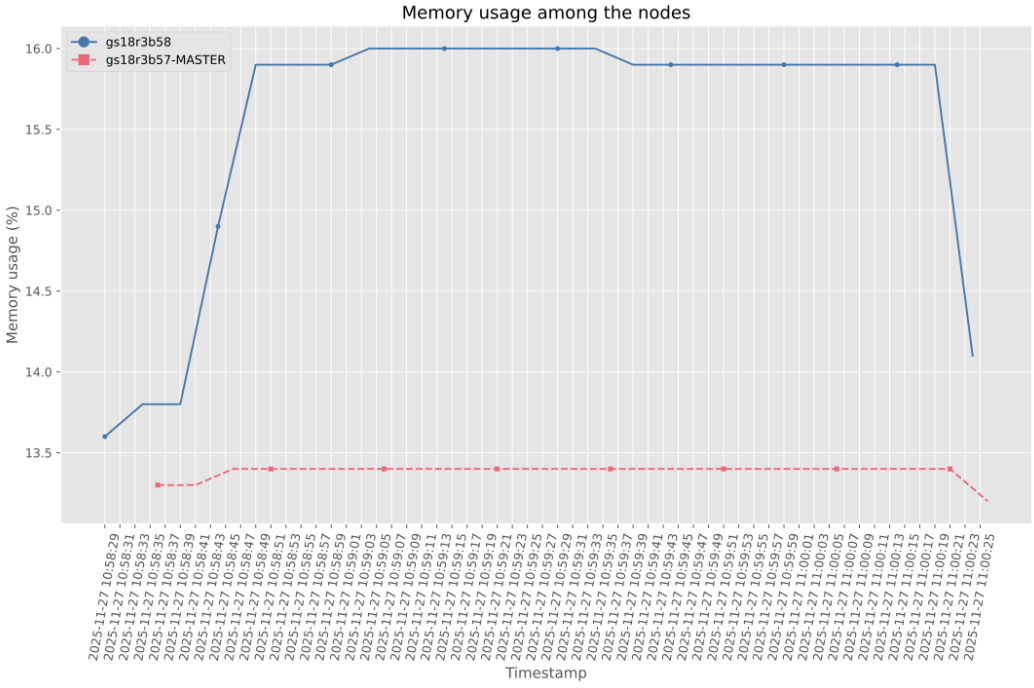
3.4 Profiling Visualization

Here it is possible to observe that different profiling tools applied on the same application provide the same results. The plots below illustrate that the performance metrics align regardless of the tool used. This consistency indicates that all the profiling tools tested (`psutil`, `top`, and `cgroup`) are reliable in the same way for this workload. The next charts display the performance profiling of a Matrix Multiplication application with MSIZE=12, BSIZE=3, and one worker node (in the chapter 4, it will be explained better).

Profiling plots of `psutil`



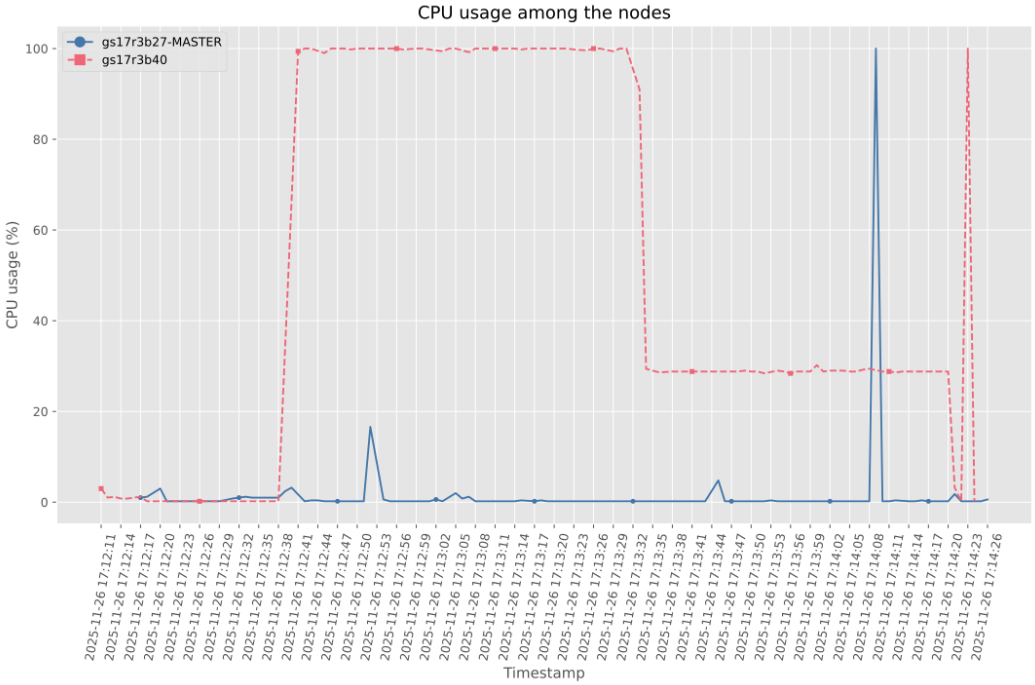
(a) CPU usage



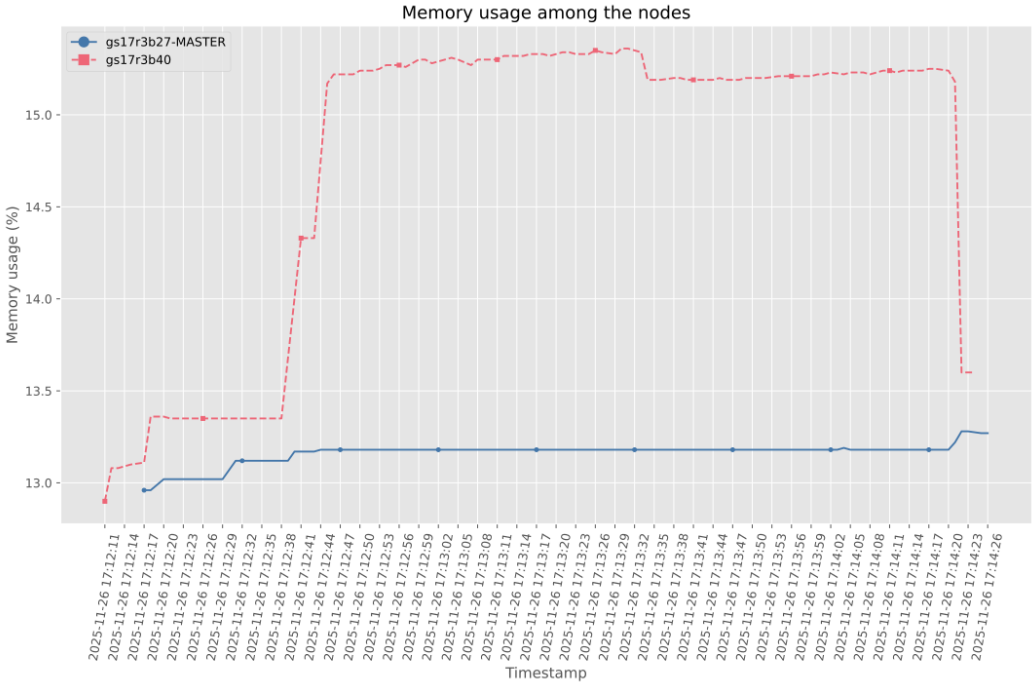
(b) Memory usage

Fig. 3.1.: Resource profiles generated using the `psutil` library.

Profiling plots of `top`



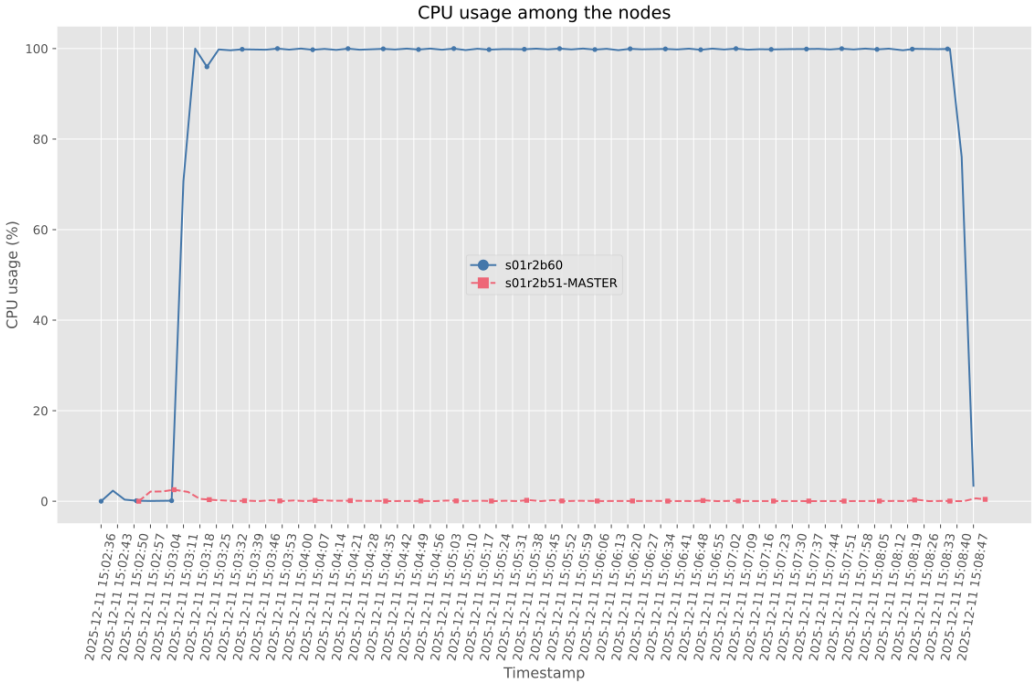
(a) CPU usage



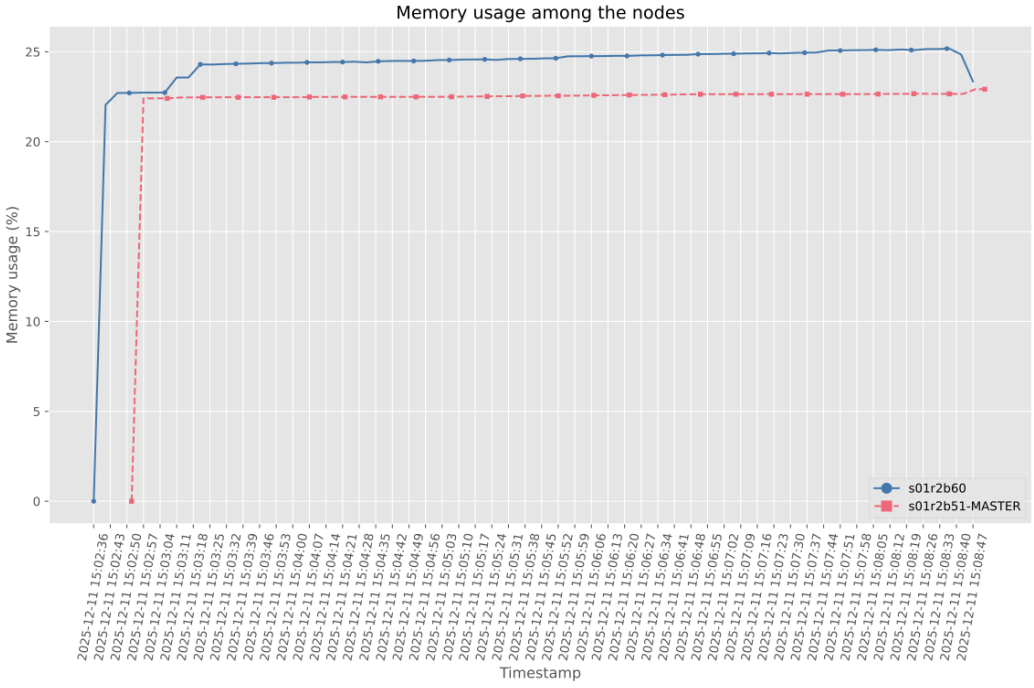
(b) Memory usage

Fig. 3.2.: Resource profiles generated using the `top` command.

Profiling plots of `cgroup`



(a) CPU usage



(b) Memory usage

Fig. 3.3.: Resource profiles generated using `cgroup` metrics. In this case it has been executed on the `Nord4`, the profiling method used on that machine

3.4.1 Discussion on the plots

Comparing them confirms that the application behavior is consistent across all monitoring methods, despite minor differences in how the data is reported.

CPU Usage Comparison

The three CPU charts illustrate the exact same execution pattern.

- **Identical Behavior**: In every chart, the Master node remains idle (near 0% usage). The active node spikes immediately when the task begins, maintains a steady level of high usage, drops to a lower level briefly, and then finishes.

- **Scale Differences:** The `top` and `psutil` graphs show the active node reaching **100%** usage. The `cgroup` graph shows the active node reaching **25%** usage. While the percentage scale differs (likely due to how cgroup counts CPU cores compared to the other tools), the **shape** of the graph is identical. This proves the tools are measuring the same workload.

Memory Usage Comparison

The memory charts (`mem_nodes-top`, `mem_nodes-psutil2`, and `mem_nodes-cgroup`) show the active node increasing its memory consumption while the master node remains stable.

- **General Trend:** All three sources show the memory rising sharply at the start of the task and holding steady until the end.

- **Data Differences:** The `cgroup` data presents slightly different values compared to `top`.

  - **top:** Shows the memory usage peaking at approximately **15.3%**. The line has small fluctuations.

  - **psutil:** Shows a peak of **16%**.

  - **cgroup:** Also shows a peak of **16%**, similar to `psutil`.

This indicates that `cgroup` and `psutil` likely track memory allocation in a similar way (possibly including cached memory), while `top` reports a slightly lower, strictly used memory figure. Despite these small numerical differences, the timing and behavior of the application are consistent across all plots.

3.5 Profiling Overhead Analysis

The profiler gets the data periodically. The user can set an environment variable by defining the interval in seconds. This allows us to set the impact of the profiling script, and it allows us to adjust the interval based on the kind of application, keeping the overhead not so intrusive on the performance of the application.

We did several tests to show the impact of the profiler on the performance of the execution in terms of time. Also, we did an analysis on the execution times for each tests to understand the variance of the times for the same execution, which will be preparatory for understanding the accuracy of the model in chapter 4.

3.5.1 Analysis of Profiler Overhead and System Stability

To validate the feasibility of using the profiler in a production environment, it is critical to assess the performance penalty introduced by the new implementation. The goal is to demonstrate that the overhead remains within acceptable limits and does not significantly alter the natural execution behavior of the application. The following analysis compares the baseline execution (No Profiler) against two profiling configurations: a high-frequency interval (1 second) and a low-frequency interval (5 seconds).

Impact on Total Execution Time

The absolute impact of the profiler on execution time is visualized in 3.4 (Profiler Overhead & Variance Analysis). The grouped bar chart presents the mean execution time for six different system configurations (varying MSIZE, BSIZE, and number of nodes).

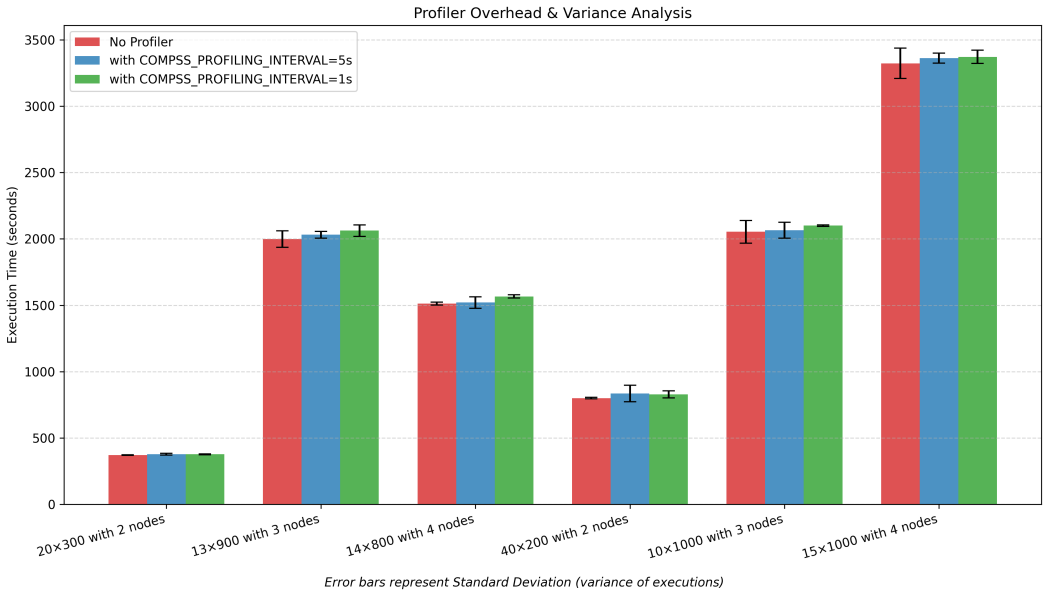


Fig. 3.4.: Execution time with different profiling interval of every test

Visually, the difference between the baseline execution (red bars) and the profiled executions (blue and green bars) is negligible. Across all tested configurations, the total execution time remains statistically consistent regardless of whether the profiler is active. For example, in the computationally intensive test with $MSIZE = 15$, $BSIZE = 1000$, and $NUM\_NODES = 4$, the execution time exceeds 3300 seconds. In this scenario, the addition of the profiling routine results in a marginal increase in time that is visually indistinguishable on the scale of the graph. This indicates that the profiler sampling mechanism is insignificant compared to the main workload of the application.

Analysis of Execution Variance

Beyond average execution time, stability is a key metric for performance analysis. 3.4 includes error bars representing the standard deviation of the executions.

The data demonstrates that the profiler maintains system stability. The variance observed in the profiled runs is comparable to, and in some cases lower than, the variance of the baseline runs. For instance, in the test with $MSIZE = 10$, $BSIZE = 1000$, and $NUM\_NODES = 3$, the baseline actually exhibits a larger standard deviation than the recorded runs. This suggests that the variations in execution time are driven by inherent system factors rather than the profiler itself. The profiler does not introduce meaningful instability or unpredictability to the runtime environment.

Quantification of Relative Overhead

To provide a precise measurement of the cost, 3.5 (Profiler Overhead Percentage Analysis) details the relative overhead percentage for each configuration.

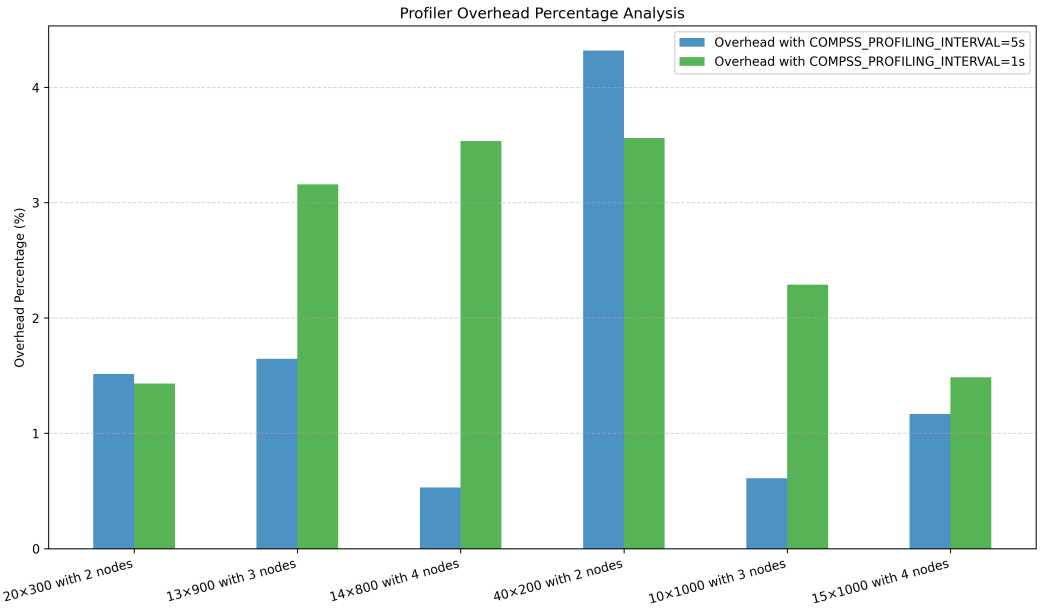


Fig. 3.5.: Overhead in percentage of different profiling interval compared to the baseline (no profiling)

The results confirm that the overhead is extremely low. Also, by the table 3.4, we can conclude with the following statements:

- **Minimal Impact**: In all cases, the overhead is below 4.5%. For the majority of configurations, the overhead for the 5-second interval is below 1%.

- **Interval Sensitivity**: As expected, the 1-second interval (Green) generally incurs slightly higher overhead than the 5-second interval (Blue) due to the increased frequency of context switches and data collection. However, even at the aggressive 1-second interval, the penalty never exceeds 4% of the total execution time.

- **Scalability**: The overhead does not appear to grow disproportionately with the problem size. Whether the task takes 300 seconds or 3000 seconds, the relative percentage remains strictly bounded.

Test	MSIZE	BSIZE	NUM_NODES
Test 1	20	300	2
Test 2	13	900	3
Test 3	14	800	4
Test 4	40	200	2
Test 5	10	1000	3
Test 6	15	1000	4

Tab. 3.2.: Test ID Mapping.

Test	AVG_NO	AVG_5	AVG_1	VAR_NO	VAR_5	VAR_1	VAR
Test 1	371.56	377.18	376.88	4.04	50.76	14.06	24.71
Test 2	1998.75	2031.63	2061.86	3975.38	662.97	1820.82	2362.03
Test 3	1512.25	1520.28	1565.70	123.06	1818.97	150.96	1146.37
Test 4	799.85	834.39	828.33	27.80	3796.63	690.61	1383.80
Test 5	2053.00	2065.50	2100.00	7298.06	3511.03	28.16	3153.81
Test 6	3322.62	3361.37	3371.93	13021.83	1441.37	2428.30	4728.51

Tab. 3.3.: Averages and Variance.

Test	STD_NO	STD_5	STD_1	%STD_N	%STD_5	%STD_1	OVHD_5	OVHD_1
Test 1	2.01	7.12	3.75	0.54%	1.89%	0.99%	1.51%	1.43%
Test 2	63.05	25.75	42.67	3.15%	1.27%	2.07%	1.65%	3.16%
Test 3	11.09	42.65	12.29	0.73%	2.81%	0.78%	0.53%	3.53%
Test 4	5.27	61.62	26.28	0.66%	7.38%	3.17%	4.32%	3.56%
Test 5	85.43	59.25	5.31	4.16%	2.87%	0.25%	0.61%	2.29%
Test 6	114.11	37.97	49.28	3.43%	1.13%	1.46%	1.17%	1.48%

Tab. 3.4.: Standard Deviation and Overhead.

3.5.2 Conclusion

The experimental data conclusively demonstrates that the developed profiler has light impact and it is non-intrusive. With a performance penalty consistently below 5%, and often below 1% for standard sampling intervals, the profiler does not significantly affect negatively the performance of the application. Furthermore, the variance analysis confirms that the tool does not degrade system stability.

3.6 Metadata Storage Architecture

Research Objects (RO-Crates) are not limited to descriptive metadata. Instead, they are self-contained bundles (typically represented as `zip` archives) that include both metadata and all the associated artifacts, such as source code, images, plots, datasets, configuration files, input/output files, and any other resources relevant for the generation of the RO-Crate object. For this reason, a storage solution that retains only the metadata would cause a substantial loss of information, since the contextual and executable components of the research objects would be omitted. Therefore, the database architecture had to be designed to include:

- Structured and standardized metadata in JSON-LD format
- Artifacts included in the RO-Crate object

During the design phase, some databases were evaluated in order to identify the most suitable solution.

3.6.1 Preliminary Evaluation: MongoDB

The first system investigated was MongoDB, a widely adopted document-oriented NoSQL database. Given that the `ro-crate-metadata.json` files are JSON documents, MongoDB initially appeared to be an appropriate candidate, as it natively supports the storage and retrieval of JSON data. Its flexibility, ease of deployment, and efficiency in handling semi-structured data are well-known advantages.

However, a critical limitation emerged: MongoDB stores arbitrary JSON documents without setting any semantic constraints. This implies that the Linked Data JSON-LD structure of the RO-Crate metadata would not be preserved. As a consequence, it is not possible to interact with the database exploiting the semantic relationships. In particular, the absence of a standardized RDF model prevents the execution of SPARQL queries. Thus, although MongoDB provides flexibility and speed, it introduces two major drawbacks: loss of semantic richness inherent in JSON-LD and incompatibility with semantic web technologies.

Advantages of MongoDB	Limitations of MongoDB
Native support for JSON documents	Lack of support for RDF and JSON-LD standards
High flexibility and schema-free data management	Inability to perform SPARQL queries
Easy to implement and scale horizontally	Potential loss of semantic information critical for interoperability
Fast query performance for general-purpose operations	

Tab. 3.5.: Advantages and limitations of MongoDB.

3.6.2 Adoption of RDF Databases: Apache Jena Fuseki

The investigation led to the identification of specialized RDF databases, among which *Apache Jena Fuseki*[2] emerged as the best candidate. Fuseki is a SPARQL server designed to store and query RDF datasets through standard SPARQL endpoints. Its architecture is compatible with the semantic representation of metadata in RO-Crates, as the `ro-crate-metadata.json` file can be directly converted into RDF graphs.

By adopting Fuseki, the metadata of research objects can be stored in a way that preserves their semantic richness, enabling queries that exploit relationships, hierarchies, and provenance links. Users and applications can execute SPARQL queries to filter, search, and reason over metadata. In this way, it ensures interoperability with the broader semantic web ecosystem.

However, Fuseki introduces an important limitation: it is tailored exclusively for the storage and querying of RDF metadata. It does not natively provide mechanisms to manage the included digital artifacts of the RO-Crates (e.g., source code files, images, datasets). As a result, Fuseki alone cannot fully address the storage requirements of RO-Crates.

3.6.3 Provenance Storage: Integrating Metadata and Artifacts

To overcome this limitation, the *Provenance Storage* project[2] was identified as an optimal solution. Developed by the Research Objects community, Provenance Storage

[2] `https://github.com/crs4/provenance-storage/`

is specifically designed to integrate semantic metadata management with object storage, as it aligns perfectly with the requirements of RO-Crates.

The architecture of Provenance Storage is based on the integration of two complementary components:

- **Apache Jena Fuseki**: serves as the metadata management layer, storing the `ro-crate-metadata.json` files in RDF format and exposing them through a SPARQL endpoint. This allows for semantic querying and interoperability with external semantic web applications.

- **MinIO**[17]: provides the object storage layer, implementing an S3-compatible backend for managing large volumes of heterogeneous digital artifacts. All files contained within an RO-Crate (datasets, code, images, input/output files) are stored in MinIO, ensuring high-performance and scalable access.

The integration of these two systems establishes a clear division of responsibilities: Fuseki guarantees semantic expressiveness and query capabilities for metadata, while MinIO ensures persistence and accessibility of the actual research artifacts. Each metadata record in Fuseki is linked to its corresponding objects in MinIO, thus maintaining the logical coherence of the RO-Crate structure. This hybrid architecture achieves a balance between semantic interoperability and large-scale data storage, fulfilling the core requirements of reproducible and FAIR (Findable, Accessible, Interoperable, Reusable) research data management.

3.6.4 Limitation of Provenance Storage

Although the *Provenance Storage* project provides an effective integration of *Apache Jena Fuseki* for metadata and *MinIO* for object storage, it presented a significant limitation in terms of usability. At the time of evaluation, interaction with the system was restricted exclusively to a command-line interface (CLI). While functional, this approach tightly coupled the CLI to the underlying database components and lacked flexibility for remote usage or integration with external applications. Consequently, this design did not permit broader adoption in scenarios that require network-based interactions, interoperability with external services, or more user-friendly modes of access.

To address this limitation, we contributed for the development of a middleware layer in the form of a REST API service. This service, implemented using *FastAPI*, decouples the CLI from direct access to the databases. In the revised architecture, all

CLI commands are routed through the API, which then communicates with Apache Jena Fuseki and MinIO.

3.6.5 Implementation Details

The contribution was formalized as a pull request to the Provenance Storage project, introducing a FastAPI service under `src/provstor_api/`[3]. The implementation can be summarized as follows:

- **New API service:** A FastAPI application was introduced with modular endpoints supporting operations such as `load`, `query`, `get`, and `backtrack`. These endpoints replicate the functionality previously implemented directly in the CLI but expose them over REST.
- **Docker integration:** A dedicated Dockerfile (`api.Dockerfile`) was created for the API container. The `docker-compose.yaml` configuration was updated to orchestrate the API, Fuseki, and MinIO services together.
- **Configuration management:** Environment variables were centralized into a `.env` file, simplifying deployment and runtime configuration. In addition, a client configuration file (`~/.config/provstor.config`) was introduced to allow users to specify the API endpoint for the CLI.
- **CLI refactoring:** The CLI (`cli.py`) was refactored to delegate all operations to the API, removing direct database interaction. Legacy files (`backtrack.py`, `get.py`, `list.py`, `load.py`, `queries.py`, `query.py`) were preserved for reference but their logic was reimplemented in the API layer under `src/provstor_api/routes/`.
- **Dependencies:** The `pyproject.toml` file was updated to include FastAPI and other necessary dependencies. For containerized deployment, a `requirements.txt` file was also added.

The updated architecture is illustrated in Figure 3.6.

[3] `https://github.com/crs4/provenance-storage/pull/17`

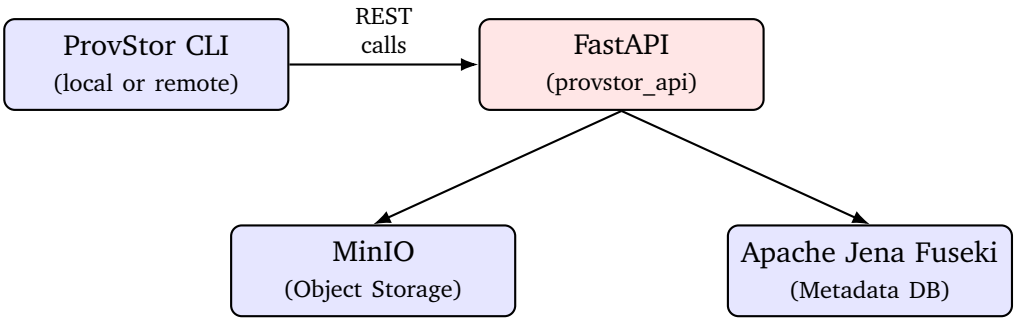


Fig. 3.6.: Extended architecture of Provenance Storage with API middleware.

Benefits of the Middleware Architecture

The introduction of the API middleware brings multiple advantages:

- **Decoupling:** The CLI no longer requires direct database access; instead, all interactions are routed through the API.
- **Remote usage:** Users can now run the CLI locally while connecting to a remote Provenance Storage server simply by updating the `provstor.config`.
- **Interoperability:** External applications can interact with the database through the REST API, extending usability beyond the CLI. This is particularly valuable for integration into automated workflows or research platforms, such as the one developed in this thesis.
- **Maintainability:** Centralization of environment variables in the `.env` file simplifies deployment, and modular API routes improve code organization.

In Figure 3.7, there is the list of the API callable function commands that are available.

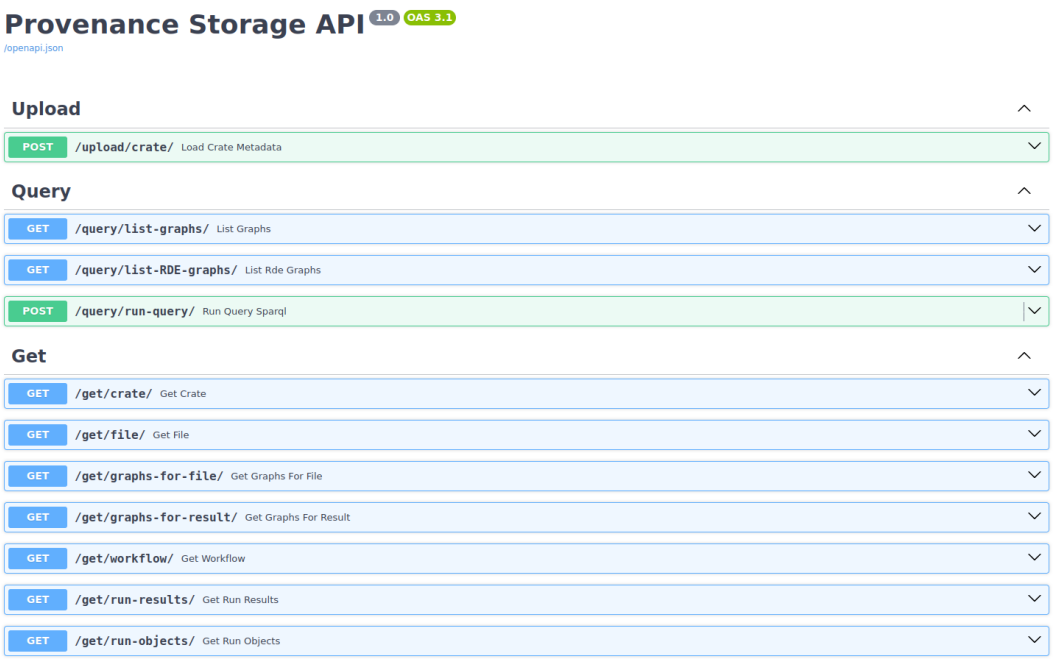


Fig. 3.7.: API callable functions in the REST API Swagger UI

4 Prediction Model

Predicting the execution time of matrix multiplication in a distributed environment is fundamentally a **Supervised Regression** problem. The task consists of training a model on a dataset of system metrics (e.g., Matrix Size, Block Size, CPU utilization) to predict a continuous numerical outcome: the execution time. This chapter details the data acquisition process, the rationale behind rejecting synthetic data, and the progressive complexity strategy employed to select the optimal machine learning architecture.

4.1 Application Chosen for the Experiments: Matrix Multiplication

Matrix Multiplication was selected as the primary application for these experiments. This algorithm is particularly suitable for benchmarking distributed systems because it offers a precise definition of every operational stage, allows for the deterministic calculation of the total number of tasks, and facilitates the distribution of workload across the system. The application exhibits a predictable scaling behavior: as the input parameters increase, the execution time rises consistently relative to the underlying architecture. For the purpose of this study, the application utilizes two primary input parameters to define the workload:

- **Matrix Size (MSIZE):** This parameter defines the overall dimensions of the matrices in terms of the file structure. In this specific implementation, the matrices are not stored as single monolithic files but are decomposed into a grid of smaller files. The MSIZE determines the number of rows and columns of this grid.

- **Block Size (BSIZE):** This parameter defines the granularity of the data within each file. Since the application utilizes square blocks, the $BSIZE$ dictates the number of rows and columns inside a single block. For example, a $BSIZE$ of 10 results in 100 items per file.

The operational workflow of the application is divided into two distinct main stages:

1. **Stage 1: Initialization of the Matrices.**
   In this phase, the matrices A, B, and C are generated. As previously noted, the data structure relies on a set of fragmented files rather than a single input. The quantity of these files is strictly determined by the MSIZE. The naming convention for these files adheres to the structure *<Name>.<Row>.<Column>*. To illustrate, if matrix A is generated with an *MSIZE*$= 10$, the system will produce 100 distinct files, from $A.0.0$ to $A.9.9$. Regarding the content, the input matrices (A and B) are initialized with item values of 2.0, while the result matrix (C) is initialized with values of 0.0.

2. **Stage 2: Execution of the Multiplication.**
   During this stage, the actual computation is performed by distributing tasks across the system. A single "task" is defined as the multiplication of one specific block. The mathematical nature of matrix multiplication allows these block computations to be highly independent. Ideally, every block computation can be executed simultaneously with others, guaranteeing a complete parallelization of the execution. This distribution is managed by the COMPSs framework, which splits the application into tasks and schedules them on different computing nodes.

The code of the application is available in Appendix B.

All the execution made for these experiments are executed on MareNostrum 5, where every node consists of 112 CPUs. Thanks to this architecture, every node can run 112 concurrency tasks. The topology employed for these experiments is the Master-Worker model. In this topology, the Master node does not execute any computational tasks; its role is strictly to initialize the matrices, distribute the data to the Worker nodes, and await the results. The Worker nodes receive the tasks, perform the block multiplications, and send the output back to the Master. Once the Master has aggregated all results, it stores the data and executes a provenance workflow to construct the Research Object Crate (RO-Crate).

4.2 Data Availability

4.2.1 Dataset Generation via Profiling

Before the implementation detailed in Chapters 3 of this thesis, a dataset suitable for this specific distributed computing environment did not exist. The profiling tool developed in the previous chapters was the instrument able to generate the training dataset. By integrating this tool into the COMPSs executions, we were able to capture system metrics during the runtime.

The dataset consists of 538 distinct execution records. These records were generated through a series of controlled experiments varying the 8 following parameters:

- **Execution time (EXEC_TIME)**: Execution time of the job in seconds
- **Matrix Size (MSIZE)**: Size of each matrix involved in the multiplication.
- **Block Size (BSIZE)**: Size of each block contained in every matrix.
- **CPUs per node (CPUS_PER_NODE)**: Number of CPU cores available per compute node. All the tests have been executed on MareNostrum 5, therefore this number is constant and set to 112 CPUs per node.
- **Number of nodes (NUM_NODES)**: Total number of compute nodes defined for the job execution.
- **Number of CPUs (NUM_CPUS)**: Total number of CPU cores allocated for the job execution.
- **CPU average (CPU_AVG)**: Average CPU utilization percentage across all allocated cores.
- **Memory average (MEM_AVG)**: Average memory utilization percentage across all allocated memory.

4.3 Data Preprocessing and Filtering

An initial analysis of the raw data revealed a high degree of heterogeneity. The experimental distribution was unbalanced, with the majority of executions involving

smaller input sizes. To address this, and to ensure the models are trained on non-trivial workloads, we applied a filter to exclude data points where the processed data size exceeded approximately 4.6 GB ($4,632,608,768$ bytes).

This filtering decision is taken because the analysis of the input size distribution along the number of nodes used was unbalanced. The boxplot of the raw dataset (Figure 4.2) exhibits extreme skewness, with outliers reaching upwards of 1.3×10^{10} bytes (approx. 12 GB), particularly visible in experiments with 5, 7, and 11 nodes. These extreme values shrink the scale of the rest of the data, and hide the average making small differences look insignificant.

Furthermore, the heatmap analysis of the raw data (Figure 4.1) confirms the analysis. The dataset is heavily dominated by "Tiny" input sizes (e.g., 86 samples for 3 nodes), while "Large" and "Huge" categories are sparsely populated (often 0 to 2 samples). This sparsity in the upper tail of the distribution introduces high leverage points that could bias predictive modeling.

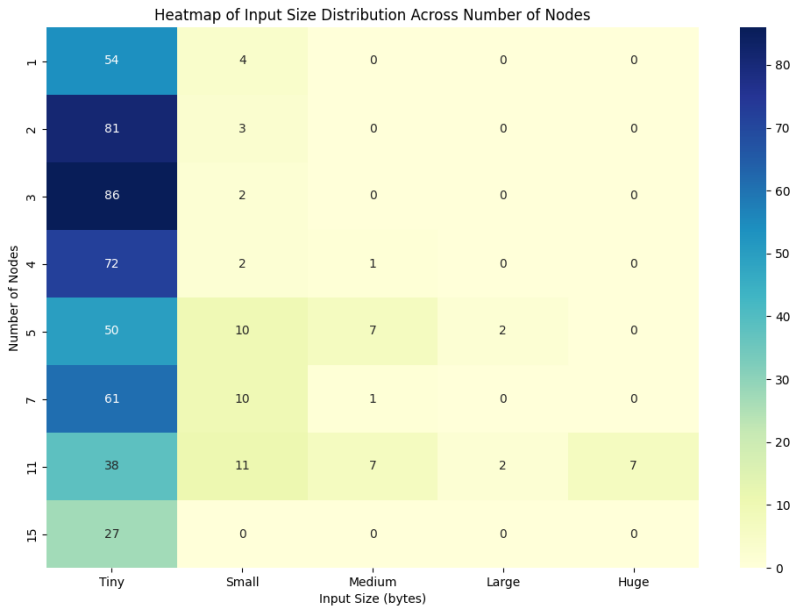


Fig. 4.1.: Distribution heatmap original dataset

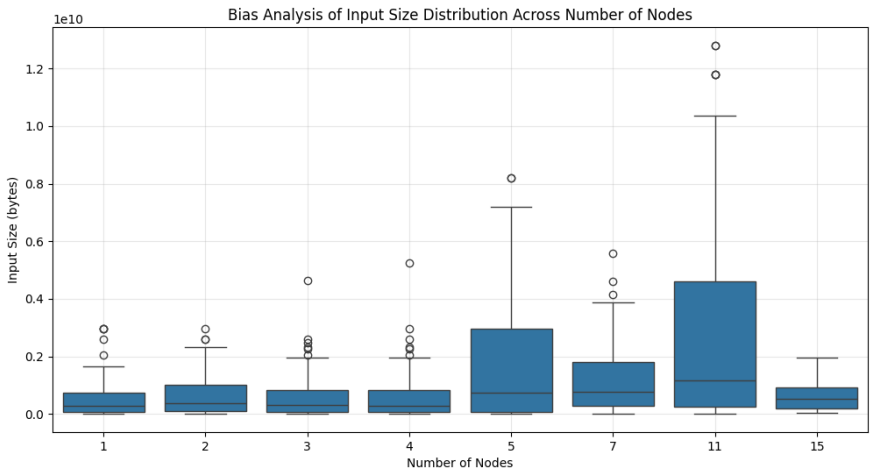


Fig. 4.2.: Distribution box plots original dataset

By applying the 4.6 GB cutoff, as shown in the filtered boxplot (Figure 4.4), we effectively truncated this long tail. The filtered distribution reveals a more consistent variance structure across node counts, exposing the median behavior that was previously minimized by the outliers. This filtering did not affect the size of the dataset, indeed the filtered dataset contains 311 records, showing that only 27 records were removed. This preprocessing step restricts the analysis to a dense, representative region of the feature space, in this way it improves the stability and generalizes the data.

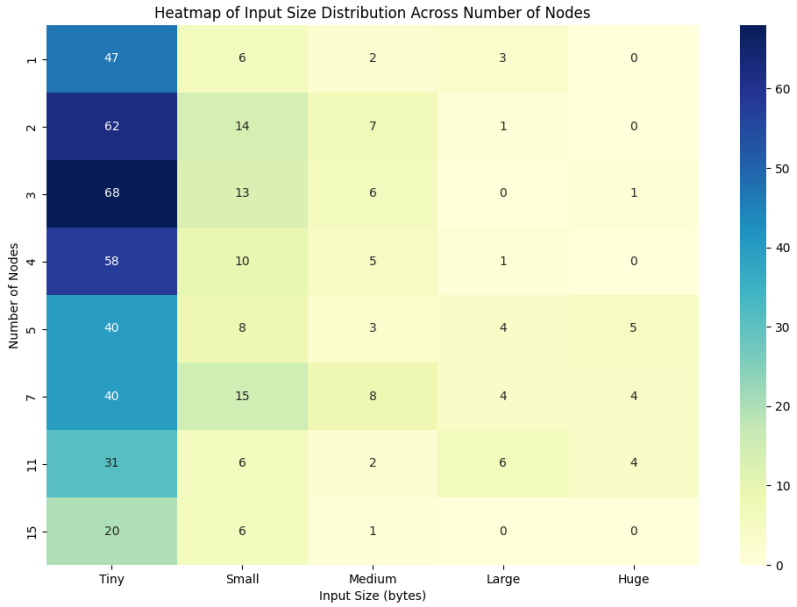


Fig. 4.3.: Distribution heatmap filtered dataset

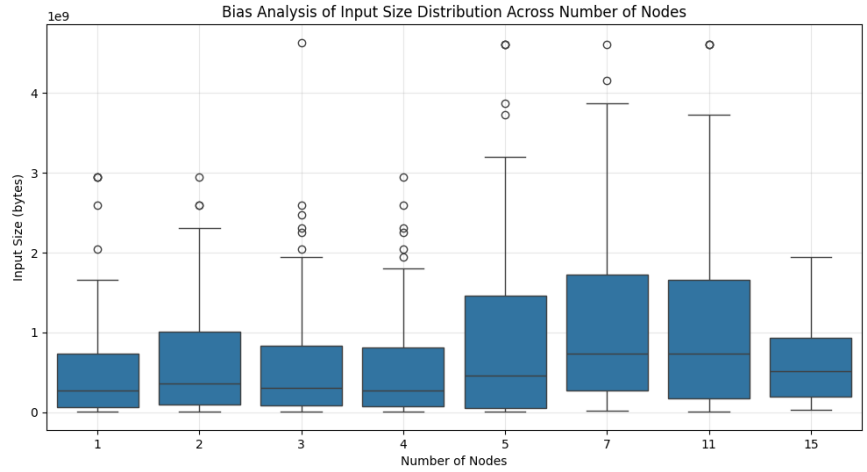


Fig. 4.4.: Distribution box plots filtered dataset

4.3.1 The Synthetic Data Problem

We deliberately avoided synthetic data generation to ensure the applicability of our results to production environments. Recent literature highlights significant discrepancies between synthetic traces and real-world HPC behavior. For instance, Guo et al. [11] argue that infusing synthetic anomalies fails to reflect real failure

patterns found in system logs. Furthermore, synthetic models frequently treat task runtimes as fixed values, ignoring the stochastic nature of data transfer and system overheads [6]. To address these limitations, our approach relies exclusively on data collected from actual executions. Thereby, we capture the realistic resource variability and patterns that synthetic models often miss.

During the early stages of this research, we considered augmenting the dataset with synthetically generated execution logs to overcome the limitation of having only 538 records. However, this approach was discarded after preliminary testing. One of the issues encountered was the determinability of synthetic patterns. Generating data that accurately reproduce the stochastic variance of a real distributed system proven to be non-trivial. Initial tests showed that the machine learning models, particularly the non-linear ones, quickly identified the function math used to generate the synthetic data rather than learning the physics of the distributed system. Given the complexity of implementing a high-fidelity simulator and the time constraints of the project, we chose to rely exclusively on the profiling data to ensure the validity of the results.

4.4 Statistical Analysis

This analysis identifies the determinants of `EXEC_TIME` using Pearson correlation (linear relationships), Spearman correlation (monotonic relationships), and Principal Component Analysis (PCA).

4.4.1 Pearson Correlation Coefficient

The Pearson correlation coefficient measures the strength and direction of a *linear* relationship between two continuous variables. It assumes that the variables are approximately normally distributed and that the relationship is linear. Values range from -1 (perfect negative linear relationship) to $+1$ (perfect positive linear relationship), with 0 indicating no linear association. Pearson correlation is particularly appropriate when changes in one variable are expected to produce proportional changes in another.

4.4.2 Spearman Rank Correlation Coefficient

The Spearman correlation coefficient measures the strength of a *monotonic* relationship between two variables, based on their ranks rather than their raw values. It does not assume linearity or normality, and it is robust to outliers. Spearman

correlation is especially useful when relationships are nonlinear but consistently increasing or decreasing.

Statistical Significance

For both correlation tests, statistical significance was assessed using a threshold of $\alpha = 0.01$. This value has been chosen to avoid the detection of false correlation. A p-value lower than 0.01 indicates that the observed correlation is unlikely to be due to random chance. However, statistical significance does not imply a strong effect size, especially in large datasets.

4.4.3 Principal Component Analysis

Principal Component Analysis (PCA) was applied to the dataset of the features in order to reduce dimensionality and understand which components have more impact on the execution time variable. PCA transforms these original variables into a smaller set of orthogonal principal components, each representing a linear combination of the original features and capturing decreasing amounts of variance. The principal components retained for analysis explain the majority of the variance in the original data.

4.5 Original Dataset

Before proceeding with the correlation analysis, it is important to define the dimensions of the dataset. The original dataset consists of **538 observations** and **7 features**. These features include workload parameters such as Matrix Size (`MSIZE`) and Block Size (`BSIZE`), hardware configurations (`NUM_CPUS`, `NUM_NODES`), and performance metrics (`EXEC_TIME`, `CPU_AVG`, `MEM_AVG`).

The primary objective of this analysis is to identify which of these features most significantly impact the execution time (`EXEC_TIME`).

4.5.1 Pearson Correlation Results

This analysis was done on the original dataset. Table C.1 in Appendix C reports the Pearson correlation coefficients between the main variables, together with their p-values and statistical significance.

The results indicate that execution time (`EXEC_TIME`) is moderately to strongly correlated with Block Size (`BSIZE`) ($r = 0.616$) and average CPU usage (`CPU_AVG`).

Matrix Size (MSIZE) also shows a significant positive correlation ($r = 0.434$), though it is weaker than Block Size. The number of CPUs exhibits only a weak linear relationship with execution time ($r = 0.149$), despite being statistically significant.

4.5.2 Spearman Correlation Results

Table D.1 in Appendix D reports the Spearman rank correlation coefficients.

Compared to Pearson, Spearman correlation reveals even stronger monotonic relationships for Block Size. The association between BSIZE and execution time is very strong (0.872), suggesting a nonlinear but strictly increasing relationship. This indicates that Block Size is a more critical determinant of performance than Matrix Size in this context.

4.5.3 Principal Component Analysis (PCA)

In this section we show the results obtained by the Principal Component Analysis (PCA) to understand which combinations of features are most strongly associated with EXEC_TIME. In this section the Principal Component Analysis was applied to the original dataset (538 observations). PCA detects the main components that are new variables able to capture the maximum variance in the dataset. These vectors are orthogonal to each other and represent directions in the data space that explain the most significant patterns or trends.

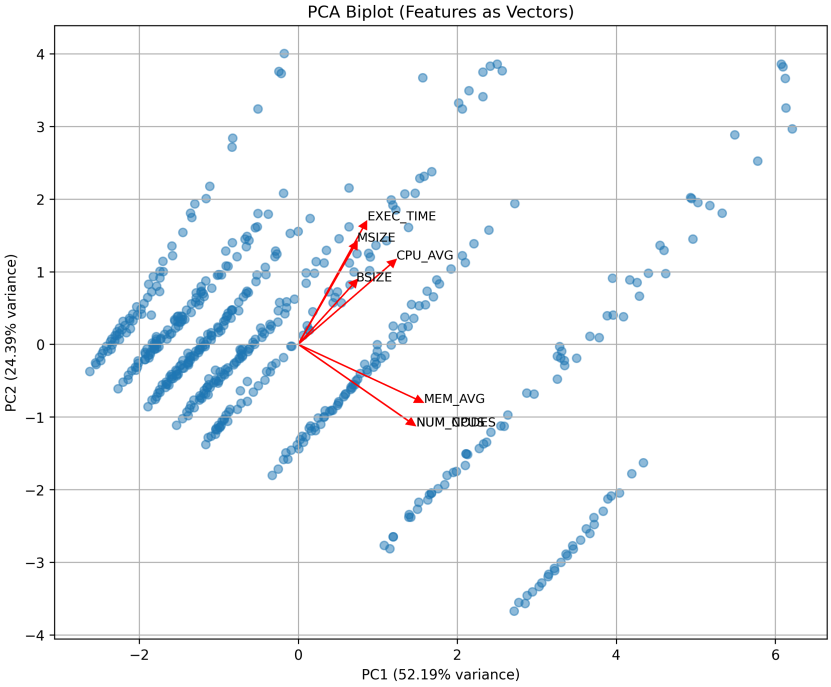


Fig. 4.5.: Biplot of the first two principal components (Original Dataset).

The first three principal components explain approximately **93.6%** of the total variance in the feature space, with PC1 alone accounting for more than half of the variability. These three components therefore capture almost all of the structure that can influence `EXEC_TIME`. The corresponding explained variance ratios are shown in Table E.1.

Table E.2 reports the loadings, which indicate how strongly each original variable contributes to each component. Focusing on `EXEC_TIME`, its loadings are moderate on PC1 (0.271), high on PC2 (0.535), and non-negligible on PC3 (0.283), meaning that the variation in execution time is mainly aligned with the second component, with additional contribution from the first and third.

Given these loadings, the main components relevant for `EXEC_TIME` can be interpreted as:

- **PC1:** Dominated by `MEM_AVG` (0.493), `NUM_NODES` (0.463), `NUM_CPUS` (0.463), and `CPU_AVG` (0.386), with a smaller but positive contribution from `EXEC_TIME` (0.271). This suggests that jobs running on larger hardware configurations and with higher average memory and CPU usage tend to have higher execution times along this hardware scale axis.

- **PC2:** Strongly associated with `EXEC_TIME` (0.535) and `MSIZE` (0.442), together with a moderate contribution from `CPU_AVG` (0.366). This component captures the dominant direction of variation in `EXEC_TIME`, indicating that larger matrices and higher CPU utilization are the main drivers of longer execution times in the original dataset.

- **PC3:** Contrasts `BSIZE` (0.716) with `MSIZE` (-0.586), while `EXEC_TIME` still has a moderate loading (0.283). This axis reflects different workload configurations (large blocks vs. large matrices), showing that execution time is also sensitive to the block size of the matrices involved in the execution.

4.5.4 Discussion and Implications

The correlation and PCA analyses highlight three key findings regarding performance drivers:

- **Block Size is the primary determinant:** `BSIZE` exhibits the strongest relationship with execution time. The gap between Pearson (r=0.616) and Spearman (r=0.872) coefficients indicates a strictly increasing, non-linear relationship. Consequently, tuning block size offers the highest potential for performance optimization.

- **Workload outweighs hardware:** `EXEC_TIME` aligns more closely with data parameters (Matrix and Block sizes) than with hardware resources. PCA results confirm that variability in execution time is driven primarily by the job characteristics (PC2 and PC3) rather than the scale of the hardware allocation (PC1).

- **Diminishing returns on scaling:** The number of CPUs and Nodes shows a weak linear correlation with execution time (r=0.149). This suggests that simply increasing hardware resources does not guarantee proportional performance gains; the specific configuration of the data workload is more critical.

4.6 Filtered Dataset Analysis

After applying data filtering procedures to remove incomplete, inconsistent, or non-informative observations, the final dataset consists of **511 observations** and **7 features**. As with the original dataset, all variables are numeric.

4.6.1 Pearson Correlation Analysis

Table C.2 in Appendix D summarizes the most relevant Pearson correlation coefficients for the filtered dataset.

The results indicate that execution time is primarily driven by Block Size (`BSIZE`), which shows the strongest linear association ($r = 0.680$). Matrix Size (`MSIZE`) shows a weaker positive correlation ($r = 0.227$).

Crucially, `NUM_CPUS` and `NUM_NODES`, show no statistically significant linear relationship with execution time ($p > 0.01$). This suggests that merely adding computational resources (CPUs/Nodes) does not linearly reduce or increase execution time in this specific workload configuration.

4.6.2 Spearman Correlation Analysis

Spearman correlation coefficients, reported in Table D.2, provide additional insight into monotonic relationships.

The Spearman analysis reinforces the dominance of Block Size (`BSIZE`), revealing a very strong monotonic relationship ($r_s = 0.877$) with execution time. This indicates that as Block Size increases, execution time consistently increases.

Unlike the Pearson results, the Spearman test detects a weak but statistically significant positive monotonic relationship between hardware resources (`NUM_CPUS`) and execution time ($r_s = 0.155$).

4.6.3 Principal Component Analysis (PCA)

PCA was also applied to the filtered dataset (511 observations) to examine whether cleaning inconsistent runs changes which components are most aligned with `EXEC_TIME`. The interpretation again focuses on how execution time projects onto the main components rather than on global variance alone.

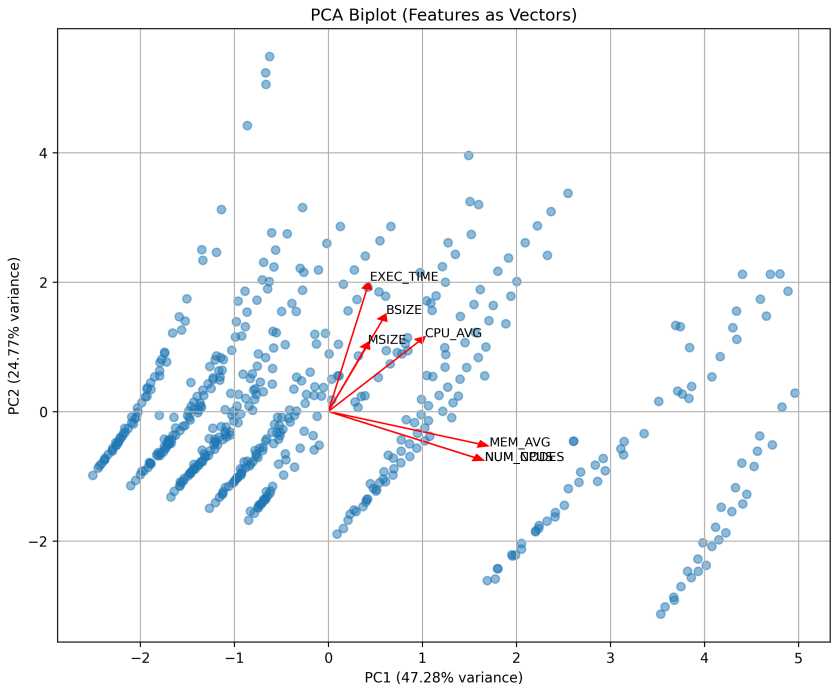


Fig. 4.6.: Biplot of the first two principal components (Filtered Dataset).

The first three components now explain approximately **92.5%** (slightly less than the original dataset) of the total variance, with PC3 taking a more prominent role (20.48%) compared to the original dataset. This indicates that the third component, which is linked to configuration effects, becomes more important after filtering. The explained variance ratios are summarized in Table E.3.

In Table E.4 we notice that the loadings for the filtered dataset show that `EXEC_TIME` aligns even more strongly with PC2 (0.633), while its loading on PC1 remains small (0.137) and on PC3 becomes negative (-0.257). This pattern suggests that the second component becomes the main axis along which execution time varies once the distribution of the dataset is aligned on all the nodes.

The filtered loading structure can then be interpreted with explicit emphasis on `EXEC_TIME`:

- **PC1:** Still dominated by `MEM_AVG` (0.534), `NUM_NODES` (0.519), and `NUM_CPUS` (0.519), with only a weak contribution from `EXEC_TIME` (0.137). This confirms that hardware scale is the largest overall source of variance, but only indirectly related to execution time after filtering.

- **PC2:** Displays a strong alignment between `EXEC_TIME` (0.633) and `BSIZE` (0.471), together with moderate contributions from `CPU_AVG` (0.361) and

MSIZE (0.327). This component is the main direction along which execution time varies, indicating that block size becomes the most important configuration parameter associated with longer or shorter execution times.

- **PC3:** The feature MSIZE (0.686) and BSIZE (-0.528) have opposite contribution, while EXEC_TIME has a moderate negative loading (-0.257). This suggests that, for a fixed EXEC_TIME level, different combinations of matrix and block size can compensate each other. As a result, this is a trade-off between total problem size and the granularity defined by the block size of the matrices.

4.6.4 Discussion and Implications

Analysis of the filtered dataset refines our understanding of performance bottlenecks:

- **Block Size is critical:** BSIZE remains the dominant predictor of execution time, showing a very strong monotonic relationship ($r_s = 0.877$). This confirms that optimizing block granularity is more effective than adjusting total problem size.

- **Hardware scaling provides no benefit:** NUM_CPUS and NUM_NODES show no statistically significant linear correlation with EXEC_TIME ($p > 0.01$). This indicates that adding resources to this specific workload yields diminishing returns or overhead rather than speed improvements.

- **Variance structure:** PCA reveals that execution time aligns strongly with PC2 (configuration parameters) rather than PC1 (hardware scale). This implies that performance depends on how the job is configured (Block Size) rather than where it runs (Node count).

4.7 Techniques and Evaluation

Evaluation Metrics

As explained, in different papers such as [22], and [25], model evaluation in execution time prediction relies on metrics such as Mean Absolute Error (MAE), coefficient of determination (R^2), and Root Mean Squared Error (RMSE) that assess how closely predicted times align with observed runtimes. In particular, in this thesis the main considered metric used to compare the performance of different machine learning is the MAE, as it measures the average magnitude of errors in their predictions without considering the direction of the error (overestimation vs. underestimation). Compared to RMSE, MAE gives a real number, so it is more intuitive than RMSE to understand how much the model is far from the real execution time. However, for completeness in every test there are all MAE, R^2 and RMSE.

Cross Validation Techniques

The training process frequently employs cross-validation techniques, which is necessary to avoid the overfitting risks. As a first attempt, the cross validation technique used is the classic *cross-data validation* technique [4]. In this case, the dataset is split in two datasets: the training dataset and the testing dataset. The training dataset is 80% of the whole dataset, and the remaining 20% of the dataset is the testing set. The dataset is shuffled before to be split in order to have diverse and unbiased data in every set.

However, since the dataset is small, another technique was implemented: the *k-fold validation* [27]. In this cross validation technique, the dataset is partitioned into k subsets and the models are trained k times, with each iteration it rotates through them using a different subset as test data and the remaining subsets for training.

4.7.1 Input of the Models

After the data analysis and considering the data available by the Matrix Multiplication application, the machine learning models implemented used mainly the three following input features:

- **Matrix Size (MSIZE)**: Size of each matrix involved in the multiplication. Based on the PCA and the correlation analysis this features has a moderate impact on the execution time. Also, it is crucial for the behavior of the system, as it defines the number of parallelizable tasks of the application.

- **Block Size (BSIZE)**: Size of each block contained in every matrix. Based on the data analysis explained previously, this is the most effective feature on the execution time. This feature defines the granulation of the problem, because it defines the the complexity of the every task executed by the system. In simple words, it defines the size of the matrix multiplication executed on every block.

- **Number of nodes (NUM_NODES)**: Total number of compute nodes defined for the job execution. Although the number of node has not statistically a big impact on the execution time, the logic of the application imposes that the execution time is inversely dependent by the number of nodes used by the execution. Ideally, more nodes the user configures to run during the execution, more CPUs are available to be used, less time the application takes.

As input, the CPU and memory usage have not been included. The choice is due to the fact the CPU and memory utilization is not a value that is possible to know before the prediction, therefore it is considered as an output of the application execution, not an input.

4.8 Choice of Machine Learning Models

In this thesis, we increased progressively the complexity of the models at every step. We began with simple implementation to capture the baseline and the dominant theoretical trend and then, we advanced with ensemble methods and deep learning models to capture the non-linear execution anomalies. The implementation utilized the Scikit-Learn library [21] for standard algorithms and PyTorch library [20] for the neural networks.

4.8.1 Linear Regression

We first implemented Linear Regression to establish a performance baseline. Since the matrix multiplication algorithm has a predictable theoretical cost, a linear correlation exists between specific feature combinations (e.g., N^3 and Time). Linear Regression assumes a straightforward relationship: $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots$. The coefficients ($\beta_i$) directly indicate the impact and directionality (positive or negative correlation) of each metric on execution time. The test executed comparing different values of the following parameter:

- **Scaler**: These adjust data ranges so large numbers do not unfairly dominate the result. We tested:

  - **Standard Scaler**: Centers data around the average
  - **Robust Scaler**: Ignores extreme outliers
  - **MinMax Scaler**: Squeezes data between 0 and 1
  - **None**: Raw data

- **Regressor**: These are modified versions of linear regression that prevent the model from becoming too complex (overfitting):
  - **Lasso**: It is a linear regression technique that adds a penalty to the loss function to prevent overfitting. This penalty is based on the absolute values of the coefficients.
  - **Ridge Regression**: It is a technique used to prevent overfitting by adding a penalty term to the loss function. In this case, the penalty is proportional to the square of the magnitude of the coefficients.

- **Alpha**: This controls the strength of the Lasso or Ridge penalty. A high Alpha forces a simpler model, while a low Alpha allows the model to fit the training data more strictly. We tried the following values: 0.01, 0.1, 1, 10, 100.

4.8.2 Polynomial Regression

While useful as a baseline, Linear Regression fails to capture the algorithmic complexity of matrix operations. Matrix multiplication is computationally cubic ($O(N^3)$). Polynomial regression aligns the statistical model with this known theoretical complexity by introducing higher-order terms (x^2, x^3). This addresses the underfitting inherent in the linear model when applied to rapidly scaling matrix dimensions. The test executed comparing different values of the following parameter:

- **Scaler**: In this case, too, the data was adjusted in ranges so large numbers do not unfairly dominate the result. We tested: **Standard Scaler**, Robust Scaler, MinMax Scaler, None.
- **Regressor**: Lasso, Ridge Regression.
- Alpha: We tested the following values: 0.01, 0.1, 1, 10, 100.

- Polynomial degree: It is the degree of the polynomial function. A higher degree may fit the data more closely but it can lead to overfitting. The degrees tested are: 2, 3.

4.8.3 Random Forest

To move beyond fixed mathematical equations, we implemented Random Forest. As a bagging ensemble, it constructs multiple decision trees during training and outputs the mean prediction. It captures sharp performance shifts, which polynomial curves miss. By averaging multiple trees, the model remains stable and ignores the noise typical of system logs [25]. The test executed comparing different values of the following parameter:

- **Number of trees**: Defines the number of decision trees in the Random Forest. Values tested: 10, 50, 100, 200, 300, 500, 1000.

- **Depth**: The maximum depth of the tree. None means no limit. Values tested: None, 10, 20, 30.

- **Splits**: The minimum number of samples required to split. Values tested: 2, 5, 10.

4.8.4 Feedforward Neural Network (FFNN)

The Feedforward Neural Networks (specifically Multilayer Perceptrons) were implemented and tested to model performance due to three key factors [12]:

- **Universal Approximation**: The selection relies on the *Universal Approximation Theorem*. This theorem states that a neural network with a single hidden layer and specific activation functions can approximate any continuous function to any desired degree of accuracy. This allows the model to learn the relationship between input parameters (MSIZE, BSIZE, number of nodes) and execution time without requiring an explicit mathematical formula derived from physical laws.

- **Non-linearity and Complexity** FFNNs effectively model complex, multi-dimensional dependencies. While linear models are limited to straight line correlations, Deep Neural Networks (DNNs) utilize activation functions (such as ReLU) within each neuron.

- **Adoption for Regression Tasks**: MLPs are a standard, validated approach for regression in High-Performance Computing (HPC). Literature confirms that they handle high-dimensional inputs efficiently, offering competitive accuracy with lower computational costs compared to complex distance-based algorithms.

The architecture of the model has been chosen is simple and it is composed by one input layer, one hidden layer and one output layer. As with only 3 input variables, the complexity of the problem likely does not justify a deep network. Adding more layers increases the parameter count, raising the risk that the model will fit the noise rather than the actual signal. Moreover, in the paper [13] they say that to sustain good generalization, the network should use the least possible number of neurons and weights required to obtain a reasonable training error. Therefore, the main idea was to build a small network and increasing the number of neuron for each layer in order to find the best results. The risk in this kind of model is to overfit very easily. Here is the explanation of the parameters and the values chosen for your Neural Network tests.

- **Maximum number of epochs**: One epoch is one full cycle through your entire training dataset. This parameter sets the maximum limit for how long the training is allowed to run. The value 1000 is a safety enough to give the model time to learn the pattern, but ensuring the program eventually stops if the model fails to converge.

- **Hidden neurons list**: This controls the width of the hidden layer. More neurons allow the model to learn more complex shapes but increase the risk of overfitting (memorizing noise). These are the values: 2, 4, 8, 16, 32, 64.

- **eta (Learning rate)**: This controls the step size the algorithm takes when adjusting weights to reduce error. Values: 0.01 and 0.001

4.8.5 XGBoost (Boosting Ensemble)

XGBoost (eXtreme Gradient Boosting) was selected as a more sophisticated alternative to Random Forest. This machine learning algorithm builds an ensemble of weak learners (typically decision trees) sequentially, where each new tree corrects the errors made by the previous ones. Each new tree is trained to predict these residuals, and when added to the ensemble, it helps correct the previous mistakes. This process continues until a stopping criterion is true. The main reason of this model implementation is that gradient boosting is widely considered one of the most suitable for tabular data, which is the type of data that we obtained by the database

interaction. Guo et al. [5, 11] note that tree-based ensemble methods are widely adopted because they outperform distance-based algorithms like logistic regression, SVM, and kNN. These ensembles are particularly effective at detecting overhead patterns, which simpler models often misinterpret as noise. We tested the following XGBoost parameters to optimize the model:

- **Evaluation metric**: This determines how the model calculates error during training. We tested RMSE and MAE metrics.

- **Max depth**: This limits how deep and complex each individual tree can grow. We tested 2, 3, and 4 to prevent the model from memorizing noise.

- **eta (Learning Rate)**: This controls how much each new tree changes the overall model. We tested 0.01, 0.05, and 0.1.

- **lambda**: This is the L2 regularization term, functioning identically to Ridge regression. Values tested: 1 and 5.

4.8.6 Excluded Models

Several popular architectures were analyzed and rejected based on the specific nature of our data:

- **Recurrent Neural Networks (RNN/LSTM):** These are designed for sequential time-series data. Our input features (MSIZE, BSIZE, Nodes) are static configuration parameters, not temporal sequences [8].

- **Convolutional Neural Networks (CNN):** CNNs rely on spatial locality (like pixels in an image). Our feature vectors lack spatial structure, making CNNs architecturally inappropriate.

- **Distance-Based Algorithms (SVR and k-NN)**: Models such as Support Vector Regression (SVR) and k-Nearest Neighbors (k-NN) have been excluded, as they both depend on distance calculations to make predictions. However, research shows that these methods don not work well with tabular datasets which contains features measured on different scales (for example, seconds vs. bytes) and different dimensions. This kind of datasets makes distance calculations less meaningful. In contrast, tree-based methods like XGBoost and neural networks (FFNN) handle this type of data much better.

4.8.7 Why not use a Graph Neural Network

Based on the inspection of the `ro-crate-metadata.json` file, the data format follows a Linked-JSON structure. Furthermore, the data stored in the Apache Jena Fuseki database adopts the Turtle (RDF) format, which can be represented as a graph. Each entry consists of a subject, predicate, and object that form a network of interconnected nodes. At first glance, such a representation may suggest that a Graph Neural Network (GNN) would be the most appropriate architecture for downstream predictive tasks. However, after a thorough analysis of the problem setting, we conclude that a GNN is not the most suitable choice for our use case [3]. The decisive factor lies in the assumption that the *topology* of the graph (i.e., its edges, neighborhoods, or walk structures) does not carry predictive information beyond what is already contained in the raw values of the nodes themselves, together with an optional global context. Another important consideration is that each graph corresponds to a single sample of the dataset (e.g., one execution), rather than all data being represented within a single unified graph. Therefore, it is more natural and effective to treat each graph as a tabular instance, where the node values constitute the features of the sample. Under these conditions, implementing a GNN would introduce unnecessary complexity without measurable predictive advantages. The final chosen strategy is to recast the problem in a tabular form, where the values of the other nodes (along with the contextual features) are directly used as inputs to the model. This approach is both simpler and more aligned with the actual data-generating process, and it offers greater interpretability and easier implementation, while still retaining strong predictive performance in expectation.

4.9 First attempt: pure Machine Learning models

4.9.1 Linear Regression

Original Dataset The linear regression is the baseline implemented method. It is useful to understand if there is a linear function that is able to determine the prediction time. By the tests made in the previous sections, our prior hypothesis is that this regression model is not able to detect the real execution time, because the linearity is not perfect, but this test helped us to understand how far the research is.

In the table 4.1, we show the best 5 results.

Scaler	Regressor	Alpha	Test MAE	Test RMSE	Test R^2
StandardScaler	Ridge	100.0	1009.87	1773.63	0.5776
MinMaxScaler	Ridge	10.0	1015.07	1839.15	0.5492
MinMaxScaler	Lasso	100.0	1016.38	1843.42	0.5484
RobustScaler	Lasso	100.0	1037.05	1735.27	0.5930
RobustScaler	Ridge	10.0	1052.03	1725.20	0.5970

Tab. 4.1.: Model Performance Metrics Across Scalers and Regularization Types (Test Set Only)

The alpha is high, confirming that the data are diverse, so the approximation is large. This is also confirmed by the fact that the R^2 is high, which is a good index to understand the discrepancy between the real execution time and the execution time and the regression formula.

Filtered Dataset After the first test with the original dataset, we tested whether there is an improvement of the results of linear regression using the so called "Filtered Dataset". As explained in the previous section, this dataset is more balanced and less biased by the outliers.

In the table 4.2, we show the best 5 results of the experiment.

Scaler	Regressor	Alpha	Test MAE	Test RMSE	Test R^2
RobustScaler	Lasso	100.0	686.31	1237.45	0.5963
StandardScaler	Ridge	100.0	686.38	1249.10	0.5880
StandardScaler	Lasso	100.0	691.91	1222.19	0.6060
MinMaxScaler	Ridge	10.0	694.59	1292.70	0.5613
RobustScaler	Ridge	10.0	708.27	1211.09	0.6091

Tab. 4.2.: Model Performance Metrics Across Scalers and Regularization Types (Test Set Only)

As we can see, the decreasing of MAE and RMSE is significant compared to the original dataset. However, the $R\check{s}$ did not improve, confirming that the model is still far from the correct prediction.

4.9.2 Polynomial Regression

Original Dataset The polynomial regression can be considered as an improvement of the linear regression. This model is useful to understand if there are different linear relations between the input and the execution time. Our prior hypothesis in this case is that there is a linear relations between the execution time and the input, in particular one of the most important relation is that the higher the number of nodes used, the lower the execution time. However, considering the analysis made on the dataset, the model should not sufficient to predict the execution time.

In the table 4.3, we show the best 5 results obtained with the original dataset.

Scaler	Degree	Alpha	Test MAE	Test RMSE	Test R^2
StandardScaler	3	10.0	365.76	780.84	0.9154
RobustScaler	3	10.0	368.92	825.22	0.9059
RobustScaler	3	1.0	377.52	763.25	0.9194
StandardScaler	3	1.0	382.47	761.13	0.9199
RobustScaler	3	0.1	386.51	761.16	0.9199

Tab. 4.3.: Test Performance of Polynomial Ridge Regression Models with Different Scalers and Regularization Strengths

The results indicate that a third-degree polynomial fits the data well, achieving a high R^2 (> 0.91). The best performance in terms of Mean Absolute Error (MAE) is achieved using 'StandardScaler' with a higher regularization strength ($\alpha = 10.0$). This suggests that while the model requires flexibility (Degree 3) to capture non-linear trends, it also benefits from strong regularization to prevent overfitting to the noise in the original dataset.

Filtered Dataset The same polynomial ridge regression analysis was applied to the filtered dataset. This step aims to verify if removing inconsistent runs allows the polynomial model to generalize better and reduce prediction errors.

In the table 4.4, we show the best 5 results of the experiment.

Scaler	Degree	Alpha	Test MAE	Test RMSE	Test R^2
RobustScaler	3	10.0	305.65	726.09	0.8578
StandardScaler	3	10.0	312.19	694.67	0.8689
RobustScaler	3	1.0	321.43	679.54	0.8708
StandardScaler	3	1.0	327.37	678.80	0.8705
RobustScaler	3	0.1	330.58	678.44	0.8702

Tab. 4.4.: Test Performance of Polynomial Ridge Regression Models (Improved Configuration)

Filtering the dataset significantly reduced the prediction error, dropping the MAE from 365.76 to 305.65. Interestingly, the R^2 score decreased slightly to ≈ 0.86. This decrease is expected when high-leverage outliers (which can inflate variance explained) are removed. Overall, the lower MAE confirms that the model predicts "normal" execution times more accurately on the filtered data. The ‘RobustScaler‘ with $\alpha = 10.0$ proved to be the most effective combination here.

4.9.3 Random Forest

Original Dataset Random Forest is an ensemble learning method that constructs multiple decision trees during training. Unlike the regression models, it does not assume a specific algebraic relationship between inputs and outputs. This makes it particularly suitable for detecting non-linear patterns without requiring feature scaling.

In the table 4.5, we show the best 5 results obtained with the original dataset.

Trees	Depth	Min Split	Test MAE	Test RMSE	Test R^2
500	None	2	389.58	1114.84	0.8388
500	30	2	389.58	1114.84	0.8388
500	20	2	389.59	1114.82	0.8388
1000	None	2	390.72	1117.21	0.8379
1000	30	2	390.72	1117.21	0.8379

Tab. 4.5.: Test Performance of Random Forest Models with Varying Hyperparameters

The Random Forest model achieves an R^2 of ≈ 0.84, which is lower than the Polynomial Ridge Regression ($R^2 \approx 0.91$). The discrepancy between the MAE (≈ 390) and the very high RMSE (> 1114) indicates that the model is heavily penalized by large errors on a the training data. Likely the decision trees capture the general trend, but they struggle to generalize on the inconsistent outliers in the original dataset.

Filtered Dataset We applied the same Random Forest configuration to the filtered dataset. In the table 4.6, we show the best 5 results of the experiment.

Trees	Depth	Min Split	Test MAE	Test RMSE	Test R^2
500	None	2	235.68	710.27	0.8669
500	30	2	235.68	710.27	0.8669
500	20	2	235.68	710.27	0.8669
300	20	2	235.78	707.75	0.8685
300	30	2	235.78	707.75	0.8685

Tab. 4.6.: Test Performance of Random Forest Models (Improved Configuration)

The results on the filtered dataset shows a big improvement in model performance. The MAE dropped by nearly 40% (from ≈ 390 to ≈ 235), and the RMSE decreased significantly (from > 1100 to ≈ 710). The R^2 improved to ≈ 0.87. The consistency of the results across different tree depths (None, 30, 20) indicates that the model is robust and that the removed data was indeed noise that previously prevented the trees from finding accurate split points.

4.9.4 K-Fold Validation

Following the methodology showed in [22], we adopted a similar validation strategy to address data constraints. Given the limited dataset size (N=538), standard train-test splits could yield biased evaluations, especially with more complex models, which are easy to overfit. For this reason, the methodology adopted for FFNN and XGBoost was K-fold cross-validation. This approach ensures that every data point contributes to both training and validation phases, providing a statistically robust estimate of model performance [24].

4.9.5 Feedforward Neural Network

4.9.6 Feedforward Neural Network

A Feedforward Neural Network (FFNN) was implemented to capture complex, non-linear dependencies between the input configuration (Matrix Size, Block Size, Hardware) and the execution time.

Unlike the polynomial or random forest approaches, the FFNN processes input features through a series of hidden layers consisting of neurons with non-linear activation functions. Information flows in one direction—from input to output—without cycles. The network learns the optimal mapping by adjusting its internal weights during training to minimize the error between the predicted and actual execution times.

Original Dataset We applied the same K-fold cross-validation approach to the filtered dataset to see if removing inconsistent runs would further refine the network accuracy.

In the table 4.7, we show the best 5 results obtained with the original dataset.

Architecture	eta	Test R^2	Test RMSE	Test MAE
64	0.01	0.9409	628.30	279.51
64	0.01	0.9337	654.30	295.50
64	0.01	0.9344	649.95	299.65
64	0.01	0.9334	669.43	308.66
32	0.01	0.9281	688.41	328.92

Tab. 4.7.: Performance of Neural Network Models with Varying Architectures and Regularization

The neural network outperforms both Polynomial Regression and Random Forest on the original dataset. It achieves an R^2 of ≈ 0.94 and a significantly lower MAE (≈ 279) compared to the previous models. This indicates that the neural network is capable of learning the complex non-linear mapping between block size, matrix size, and execution time, even in the presence of noise.

Filtered Dataset We train and evaluated the FFNN with the filtered dataset to see if removing inconsistent runs would further refine the neural network accuracy.

In the table 4.8, we show the best 5 results of the experiment.

Architecture	eta	Test R^2	Test RMSE	Test MAE
64	0.01	0.9509	410.69	195.81
64	0.01	0.9418	439.23	201.13
64	0.01	0.9319	471.87	216.36
64	0.01	0.9383	460.86	232.83
32	0.01	0.9279	485.48	243.07

Tab. 4.8.: Performance of Neural Network Models (Improved Configuration)

The filtered dataset yields the best results observed across all experiments. The MAE drops below 200, and the R^2 reaches ≈ 0.95. The substantial reduction in RMSE (from ≈ 628 to ≈ 410) confirms that the noise in the original dataset was problematic for the model precision. With clean data, the Feedforward Neural Network provides the most accurate prediction of execution time.

4.9.7 XGBoost

Original Dataset XGBoost (Extreme Gradient Boosting) is a powerful ensemble method that builds decision trees sequentially rather than independently. Unlike Random Forest, each new tree in XGBoost is trained to correct the errors made by the previous trees. This iterative approach allows the model to capture complex patterns and often produces superior results on structured data.

In the table 4.9, we show the best 5 results obtained with the original dataset.

Metric	Eta	Max Depth	Sub-sample	Lambda	Test R^2	Test RMSE	Test MAE
rmse	0.1	4	0.8	1	0.9251	729.10	255.74
mae	0.1	4	0.8	1	0.9241	733.37	256.46
mae	0.1	4	0.7	1	0.9191	760.72	270.17
mae	0.05	4	0.7	1	0.9204	756.51	270.25
rmse	0.1	4	0.7	1	0.9178	768.65	276.02

Tab. 4.9.: XGBoost Hyperparameter Configurations and Cross-Validated Performance

XGBoost demonstrates very good robustness on the original dataset. It achieves a Test MAE of ≈ 255, which is significantly better than the Random Forest (≈ 390) and even better than the Feedforward Neural Network (≈ 279). The high R^2 (0.925) confirms that the gradient boosting approach effectively recognize the noise present in the raw data better than other models.

Filtered Dataset We applied XGBoost to the filtered dataset to determine if removing inconsistencies would further enhance prediction accuracy.

In the table 4.10, we show the best 5 results of the experiment.

Metric	Eta	Max Depth	Sub-sample	Lambda	Test R^2	Test RMSE	Test MAE
mae	0.05	4	0.8	1	0.9147	516.97	166.53
mae	0.05	4	0.7	1	0.9130	527.29	167.30
mae	0.1	4	0.8	1	0.9115	526.44	168.70
mae	0.1	4	0.7	1	0.9104	535.53	171.01
rmse	0.05	4	0.7	1	0.9092	535.46	174.04

Tab. 4.10.: XGBoost Performance with Lower Learning Rate ($\eta = 0.05$) and Varying Subsampling

On the filtered dataset, XGBoost delivers the best performance of all models tested in this first part of the study. The MAE drops to ≈ 166, which is notably lower than the Neural Network result (≈ 195). Furthermore, the model

benefits from a lower learning rate ($\eta = 0.05$) on the clean data. This may be because a slower, more granular update process allows a better generalization once the noise is removed.

4.9.8 Comparison of First Stage Results

This section compares the performance of the trained models on the test set. The metrics focus on **Percentage Error** to evaluate the relative accuracy of predictions across different scales of execution time.

Original Dataset

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
Linear Regression	474.13	142.53	42.67	1221100.70	1105.03	1.20	10888.66
Polynomial Regression	105.40	29.50	13.00	47525.45	218.00	0.16	2680.00
Random Forest	14.64	4.62	3.79	831.28	28.83	0.01	282.28
FFNN (Best)	38.05	10.82	3.80	6241.81	79.01	0.02	647.66
XGBoost	**11.50**	4.74	0.12	538.88	23.21	0.00	320.37

Tab. 4.11.: Error Metrics – Original Dataset

The testing results on the original dataset show a hierarchy in model performance:

- **Tree-based models dominate:** XGBoost achieves the lowest average percentage error (**11.50%**) and the lowest variance, proving it is the most robust against the noise in the raw data. Random Forest follows with 14.64%.
- **Regression fails:** Linear Regression is completely ineffective (474% error), confirming the problem is strictly non-linear. Polynomial Regression improves this but remains unreliable (105% error).
- **Neural Network limits:** While better than regression, the FFNN (38.05%) is worse than the ensemble methods, likely due to the difficulty of converging on the global optimum with inconsistent data points.

In Figure 4.7 and Figure F.1e we can see respectively the plot of the Linear Regression and the XGBoost (the worst and the best model). In the F.1 there are all the plots obtained by this first study.

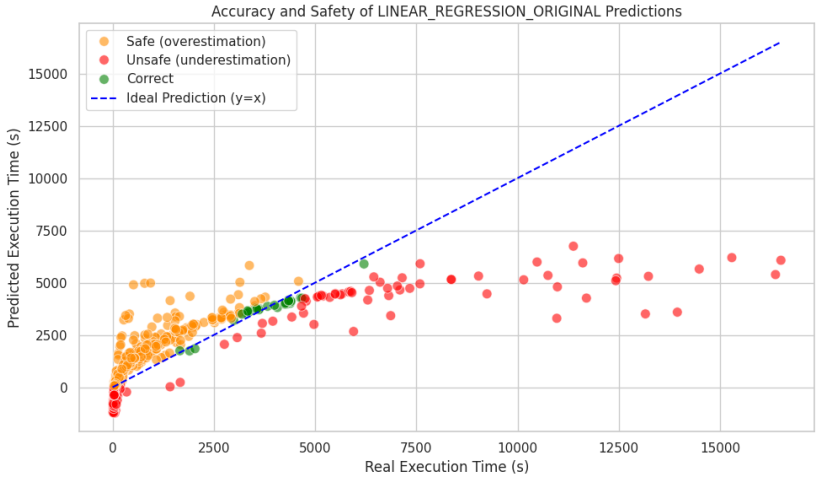


Fig. 4.7.: Plot of the prediction on the Original Dataset of the Linear Regression

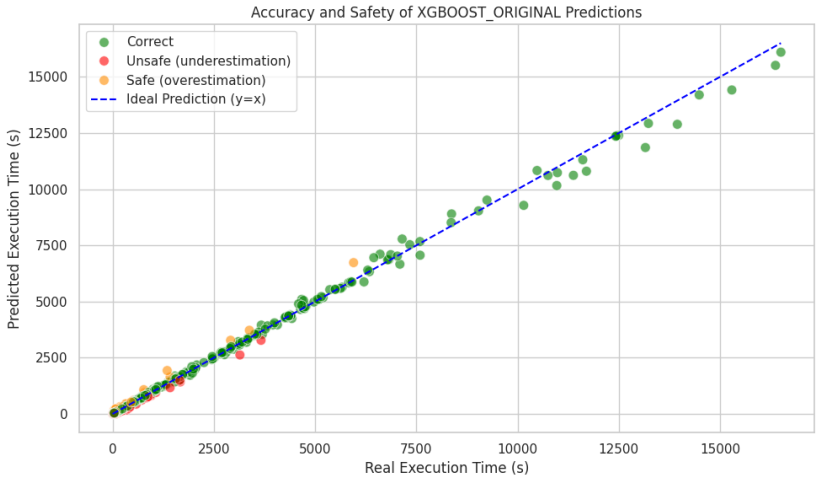


Fig. 4.8.: Plot of the prediction on the Original Dataset of the XGBoost

Filtered Dataset

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
Linear Regression	590.71	135.95	290.87	1999171.04	1413.92	1.27	13597.55
Polynomial Regression	90.83	33.77	15.99	27218.22	164.98	0.00	2136.48
Random Forest	16.23	5.15	0.35	790.43	28.11	0.04	262.26
FFNN (Best)	51.92	31.55	2.67	3050.27	55.23	0.09	302.15
XGBoost	**14.12**	4.90	0.06	772.33	27.79	0.00	384.74

Tab. 4.12.: Error Metrics – Filtered Dataset

The results on the filtered dataset confirm the superior stability of gradient boosting:

- **XGBoost maintains the lead:** It remains the best performer with an average error of **14.12%**. Although the percentage error is slightly higher than in the original dataset, it still offers the best precision.

- **Stability:** While the FFNN showed lower absolute errors (MAE) in the previous training section, its percentage error here (51.92%) suggests it struggles with relative accuracy on smaller execution time values.

We can conclude that XGBoost provides the best trade-off between bias and variance, consistently delivering errors below 15% regardless of the dataset version.

In Figure 4.9 and Figure F.2e we can see respectively the plot of the Linear Regression and the XGBoost (the worst and the best model). In the F.2 there are all the plots obtained by this first study.

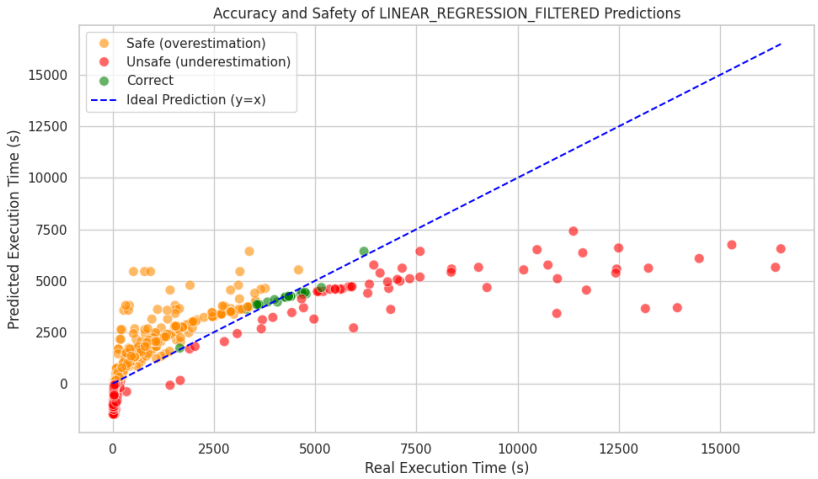


Fig. 4.9.: Plot of the prediction on the Filtered Dataset of the Linear Regression

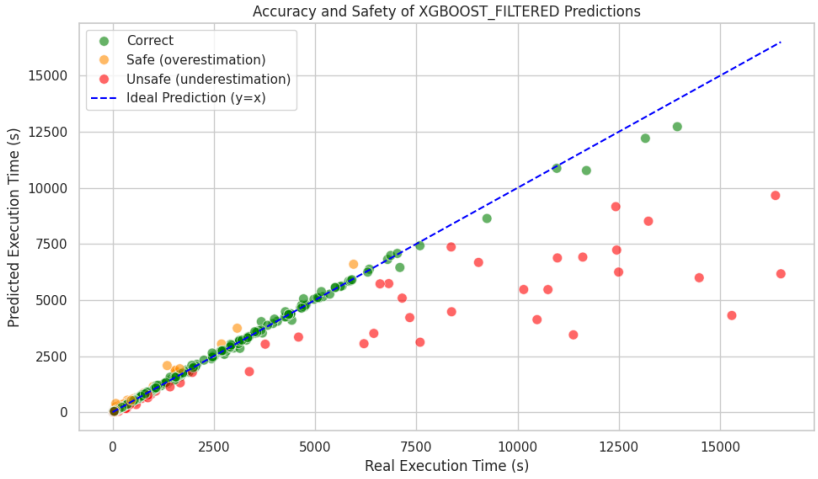


Fig. 4.10.: Plot of the prediction on the Filtered Dataset of the XGBoost

4.10 Improvements on the complex models

4.10.1 Asymmetric Objective Function

Standard regression models typically optimize symmetric loss functions, such as Mean Squared Error (MSE) or Mean Absolute Error (MAE). These functions treat positive and negative errors equivalently, so a prediction that overestimates system capacity is penalized identically to one that underestimates it.

Errors are fundamentally asymmetric in their impact:

- **Overestimation of Time (Pessimism):** Predicting that a task will take longer than it actually does leads to conservative resource provisioning. While this may incur a marginal cost in unused resources, it guarantees system stability.

- **Underestimation of Time (Optimism):** Predicting that a task will finish quickly when it is actually slow is catastrophic. It leads to potential timeouts, and job failures in the scheduler.

To address this, we introduce a custom Asymmetric Objective Function. This function imposes a heavier penalty on errors that violate safety constraints, effectively forcing the model to learn a "prudent" policy.

Mathematical Derivation

We define the custom loss function $L(y, \hat{y})$ as a weighted squared error, where the weight is conditioned on the sign of the residual. We use the L2 square error to penalize strongly huge mistakes, while it keeps being more easy to derivate (as we can see in the next paragraph). Let y be the actual residual (ground truth) and $\hat{y}$ be the predicted residual. The loss is given by:

$$L(y, \hat{y}) = w(\delta) \cdot (y - \hat{y})^2 \tag{4.1}$$

Where $\delta = y - \hat{y}$ is the prediction error, and the weighting function $w(\delta)$ is defined as:

$$w(\delta) = \begin{cases} \phi & \text{if } y > \hat{y} \text{ (Underestimation)} \\ 1.0 & \text{otherwise} \end{cases} \tag{4.2}$$

Here, $\phi > 1.0$ is the penalty factor (distinct from the USL parameter α) that controls the degree of risk aversion. When the actual time exceeds the predicted time ($y > \hat{y}$), the error is multiplied by ϕ, forcing the gradient descent to prioritize correcting these unsafe predictions.

4.10.2 Implementation in Learning Algorithms

Gradient Boosting (XGBoost)

Gradient boosting algorithms optimize the model by iteratively adding weak learners that minimize the second-order Taylor expansion of the loss function[1]. This requires the explicit calculation of the first derivative (Gradient, g) and the second derivative (Hessian, h) with respect to the prediction $\hat{y}$.

For our asymmetric objective, the derivatives are derived as follows:

$$g = \frac{\partial L}{\partial \hat{y}} = -2 \cdot w(\delta) \cdot (y - \hat{y}) \tag{4.3}$$

$$h = \frac{\partial^2 L}{\partial \hat{y}^2} = 2 \cdot w(\delta) \tag{4.4}$$

Using custom derivatives changes the gradients of the loss function. This ensures the optimization algorithm adjust stronger the correction when it encounters "unsafe" errors.

Feed-Forward Neural Networks (FFNN)

In the deep learning implementation, the asymmetric loss is integrated directly into the training loop via the PyTorch framework. Unlike gradient boosting, which requires manual derivation of the Hessian, neural networks utilize automatic differentiation to compute the gradients of the custom loss function.

The loss for a given batch is computed as the mean of the weighted squared differences:

$$\mathcal{L}_{batch} = \frac{1}{B} \sum_{i=1}^{B} w_i \delta_i^2, \quad \text{where } w_i = \begin{cases} \phi & \text{if } \delta_i > 0 \\ 1.0 & \text{otherwise} \end{cases} \tag{4.5}$$

During the backpropagation phase, the computational graph automatically propagates the higher error signal generated by the penalty factor ϕ. This effectively scales the learning rate for "unsafe" samples, causing the network weights to adjust more aggressively to correct underestimations while maintaining standard convergence behavior for safe predictions.

[1] This is an example provided by XGBoost: Custom Objective Function Tutorial.

4.11 Theoretical Enhancement: Universal Scalability Law

4.11.1 Formulation

To improve the accuracy of the scalability in a distributed system, we adopt the Universal Scalability Law (USL). The USL was formalized by Dr. Neil Gunther [10]. Unlike simple linear extrapolation or Amdahl's Law [1], which only accounts for serialization, the USL provides a unified framework that incorporates the costs of inter-process communication and data coherency. This is crucial for modern distributed architectures where cross-talk overhead significantly impacts performance at scale.

The capacity $C(p)$, representing the predicted speedup with p physical processors (or concurrent nodes), is defined by the rational function:

$$C(p) = \frac{p}{1 + \alpha(p-1) + \beta p(p-1)} \tag{4.6}$$

Where:

- p is the number of concurrent agents. In our experiment p is the number of nodes used during the execution.
- α (Alpha) represents the *contention* parameter. It models the serialization delay caused by queuing for shared resources (e.g., I/O bottlenecks). This term dominates at lower concurrency levels.
- β (Beta) represents the *coherency* parameter. It describes the latency penalty incurred by the need to maintain consistent system state across distributed nodes. This term scales quadratically with p and is responsible for *retrograde throughput*, where adding resources effectively degrades performance.

4.11.2 Implementation and Parameter Estimation

In our proposed Gray-Box modeling approach, the USL serves as the deterministic physical baseline. In practice, rather than training a machine learning model on raw execution times, we first capture the system scaling by the Equation 4.6 to the empirical data.

To determine the system specific scalability characteristics, we utilize a non-linear least squares optimization method (specifically the Levenberg-Marquardt algorithm implementation in `scipy.optimize.curve_fit`) to estimate the optimal coefficients $\hat{\alpha}$ and $\hat{\beta}$ that minimize the residuals between the observed speedup and the theoretical model. The estimation process is driven by the `usl_law` function, which accepts three fundamental variables:

- **Nodes (N)**: The number of computational resources or physical nodes utilized.
- **Observed Speedup (S_{obs})**: The empirical speedup measured from the system logs.
- **Baseline (T_1)**: The execution time required for the workload to complete on a single node.

We remind that the *Observed Speedup* is calculated using the following relationship:

$$S_{obs} = \frac{T_1}{T_N} \tag{4.7}$$

where T_1 is the `baseline_time` and T_N (referred to in the implementation as `normalized_time`) represents the execution duration recorded when the workload is distributed across N nodes.

This process separates predictable hardware limits from random workload variations. The theoretical model ensures predictions stay within the physical capabilities of the system. The machine learning model then focuses on predicting the difference between this theoretical baseline and the actual recorded times (residuals).

4.11.3 Experimental Configurations

In this tests, the the models were trained using an asymmetric loss function designed to penalize underestimation more heavily than overestimation. To evaluate the impact of the physics-informed approach, we conducted four experimental tests following a progressive research path:

Test 1 : In this test the inputs used are number of nodes and the input size calculated by MSIZE and BSIZE, which is used for the calculation of the USL formula. For this case both original and filtered dataset are tested. The physical baseline calculated in this case, follows this formula:

$$\text{baseline} = \frac{1}{N}\sum_{i=1}^{N}\frac{\text{EXEC\_TIME}_i}{\text{INPUT\_SIZE}_i} \tag{4.8}$$

Test 2 : In this case the inputs of the models are MSIZE, BSIZE, and number of nodes. Also, in this case both filtered and original dataset are tested. In this case the formula to calculate the baseline is the formula 4.8.

Test 3 : In this case, instead of the input size, the calculation of the parameters for USL formula uses the MSIZE. The formula for the calculation of the baseline is:

$$\text{baseline} = \frac{1}{N}\sum_{i=1}^{N}\frac{\text{EXEC\_TIME}_i}{\text{MSIZE}_i} \tag{4.9}$$

This change has been made because we considered that MSIZE is the number which determines the number parallel tasks to distribute in the system. In this scenario the inputs of the models are MSIZE, BSIZE, and number of nodes. The datasets tested are both original and filtered dataset.

Test 4 : In this test, to achieve another improvement, we tested the model in the Test 3 using the k-fold cross validation.

4.11.4 First attempt with USL integration (Test 1)

The inputs provided to the model were the number of nodes and a composite "Input Size" derived from `MSIZE` and `BSIZE`. The baseline for comparison is a standard Universal Scalability Law (USL) model.

The calculation of “Input Size” is the following:

$$\text{INPUT\_SIZE} = 4 \cdot (\text{BSIZE}^2) \cdot (\text{MSIZE}^2) \cdot 2 \tag{4.10}$$

The formula represents the total amount of byte in input calculated by the application, where 4 coefficient indicates the amount of byte for a float number, and 2 coefficient indicates that the initialized matrix are 2. MSIZE and BSIZE are at the power of 2 because they rapresent the dimensions of the square matrix.

Calculation of the theoretical model

By the calculation of the theoretical model, we obtain the results in Table 4.13 for the original dataset, and the results in Table 4.14 for the filtered dataset.

Parameter	Symbol	Value	Description
Serialization	α	0.3046	Contention overhead
Coherency	β	0.0229	Crosstalk overhead
Baseline Time	T_1	1.7660×10^{-6}	Single-node execution time (s)

Tab. 4.13.: Estimated USL Model Parameters for the Original Dataset

Parameter	Symbol	Value	Description
Serialization	α	0.2844	Contention overhead
Coherency	β	0.0253	Crosstalk overhead
Baseline Time	T_1	1.7659×10^{-6}	Single-node execution time (s)

Tab. 4.14.: Estimated USL Model Parameters for the Filtered Dataset

In the plot in Figure 4.11 for the original dataset and in Figure 4.12 for the filtered dataset. We can see what is the movement of the USL function based on the number of nodes used.

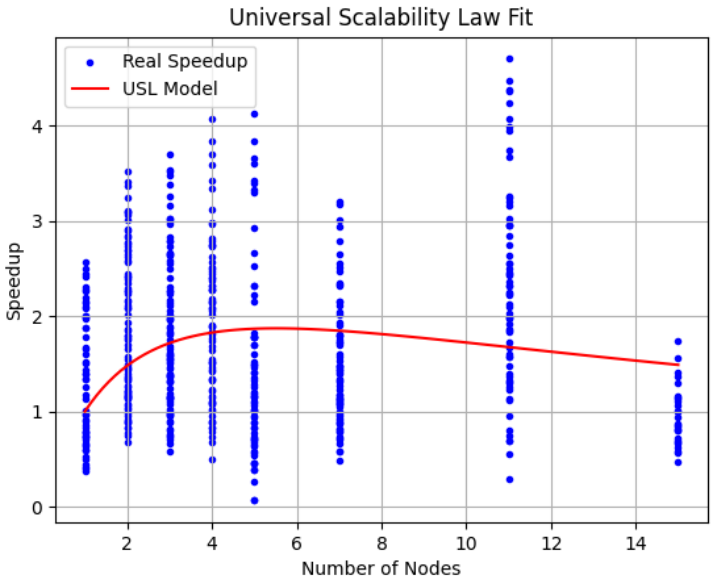


Fig. 4.11.: Plot of the USL function on the Original Dataset

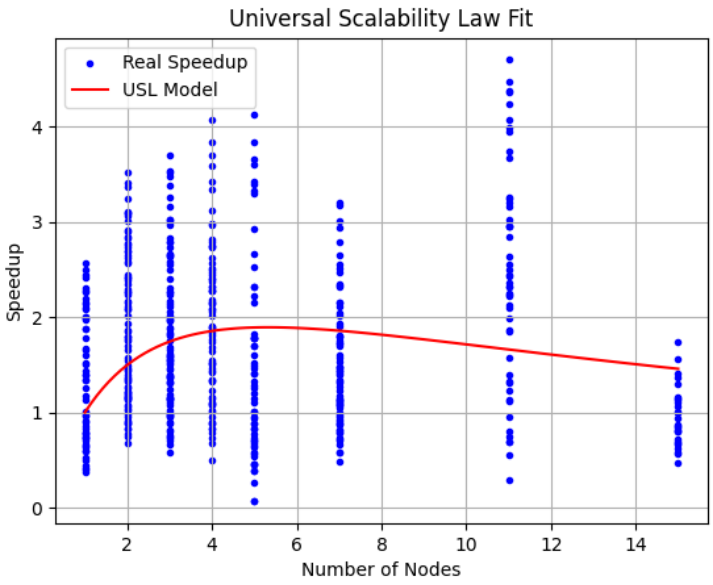


Fig. 4.12.: Plot of the USL function on the Filtered Dataset

FFNN (Original Dataset)

The results in Table 4.15 demonstrate that the hybrid neural network significantly outperforms the analytical USL model. The USL baseline yields a Mean Absolute Error (MAE) of ≈ 553. In contrast, the best FFNN configuration (Architecture 2-64-64-1 with Sigmoid activation) reduces this error to ≈ 314, representing an improvement of over **43%**. The high R^2 values ($\approx 94\%$) indicate that the neural network successfully captures the non-linear deviations that the rigid mathematical formula of the USL fails to model.

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	200	0.1	Sigmoid	2.0	553.23	996.33	314.81	538.32	43.10	93.93
2-32-32-1	150	0.1	ReLU	2.0	553.23	996.33	321.71	533.77	41.85	94.03
2-4-4-1	1200	0.01	Tanh	2.0	553.23	996.33	326.30	535.24	41.02	94.00
2-32-32-1	500	0.01	Sigmoid	2.0	553.23	996.33	327.88	534.30	40.73	94.02
2-64-64-1	500	0.1	Sigmoid	2.0	553.23	996.33	328.31	565.53	40.66	93.30

Tab. 4.15.: Performance of Feedforward Neural Networks with Asymmetric Loss and Varying Architectures

FFNN (Filtered Dataset)

The same architecture was applied to the filtered dataset to assess the impact of removing inconsistent experimental runs. As expected, the baseline USL error dropped from ≈ 553 to ≈ 465, confirming that the analytical law fits better when noise is removed.

However, Table 4.16 shows that the relative improvement of the hybrid model is smaller ($\approx 34\%$) compared to the original dataset ($\approx 43\%$). This suggests that the "Input Size" used in Test 1 may be oversimplifying the problem. By condensing `MSIZE` and `BSIZE` into a single input, we lose granularity of the problem. While the model still improves the USL baseline (MAE ≈ 303 vs ≈ 465), the lower R^2 values ($\approx 90 - 91\%$) compared to the previous test imply that the filtered data contains hidden patterns that a model with only number of nodes and input size cannot fully understand.

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-16-16-1	900	0.1	ReLU	2.0	464.79	870.89	303.39	476.14	34.72	91.15
2-4-4-1	800	0.1	Tanh	2.0	464.79	870.89	307.91	500.10	33.75	90.24
2-64-64-1	100	0.01	ReLU	2.0	464.79	870.89	308.11	484.50	33.71	90.84
2-4-4-1	1500	0.1	Sigmoid	2.0	464.79	870.89	309.79	478.97	33.35	91.05
2-64-64-1	200	0.1	Tanh	2.0	464.79	870.89	316.84	511.36	31.83	89.80

Tab. 4.16.: Performance of Feedforward Neural Networks (Second Configuration Set)

XGBoost (Original Dataset)

In this test, XGBoost was trained using the same input configuration as FFNN and an asymmetric objective function.

Table 4.17 shows that the gradient boosting approach yields a performance improvement nearly identical to the neural network. The MAE is reduced from the USL baseline of 553.23 to **318.37**, an improvement of **42.45%**. The stability of the results across different early stopping thresholds (from 50 to 500) indicates that the model converges quickly within the first 50 boosting rounds.

While the R^2 (93.00%) is slightly lower than the best FFNN result (93.93%), XGBoost demonstrates that tree-based ensembles can effectively correct the analytical USL model, even when using the simplified "Input Size" feature rather than raw matrix dimensions.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	6	50	50	3.0	553.23	996.33	318.37	620.94	42.45	93.00
mae	0.1	6	50	100	3.0	553.23	996.33	318.37	620.94	42.45	93.00
mae	0.1	6	50	150	3.0	553.23	996.33	318.37	620.94	42.45	93.00
mae	0.1	6	50	200	3.0	553.23	996.33	318.37	620.94	42.45	93.00
mae	0.1	6	50	500	3.0	553.23	996.33	318.37	620.94	42.45	93.00

Tab. 4.17.: Performance of XGBoost with Asymmetric Loss (Max Depth = 6, Penalty = 3.0)

XGBoost (Filtered Dataset)

Applying XGBoost to the filtered dataset yields the lowest error rates observed in Test 1.

The model reduces the MAE to **281.17**, surpassing both the USL baseline (464.79) and the Feedforward Neural Network (303.39). This improvement of **39.51%** confirms that while the analytical USL model benefits from clean data. Notably, XGBoost outperforms the neural network in this specific test. This suggests that for the simplified feature set, the tree-based ensemble generalizes better than the deep learning approach.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	4	100	50	2.0	464.79	870.89	281.17	520.88	39.51	91.71
mae	0.1	4	100	100	2.0	464.79	870.89	281.17	520.88	39.51	91.71
mae	0.1	4	100	150	2.0	464.79	870.89	281.17	520.88	39.51	91.71
mae	0.1	4	100	200	2.0	464.79	870.89	281.17	520.88	39.51	91.71
mae	0.1	4	100	500	2.0	464.79	870.89	281.17	520.88	39.51	91.71

Tab. 4.18.: Performance of XGBoost with Asymmetric Loss (Max Depth = 4, Penalty = 2.0)

4.11.5 MSIZE and BSIZE as inputs of the model (Test 2)

In Test 2, the model inputs were expanded to include `MSIZE` and `BSIZE` explicitly, rather than aggregating them into a single "Input Size" value. The USL formula still uses the calculated input size for its baseline prediction, but the models have now visibility into the specific matrix dimensions.

FFNN (Original Dataset)

Table 4.19 reveals a massive performance enhancement compared to Test 1. By providing the neural network with granular access to `MSIZE` and `BSIZE`, the MAE drops to **111.13**, representing an improvement of nearly **80%** over the USL baseline.

This result confirms that the relationship between workload and execution time is not solely defined by the total problem size (as assumed by the USL) but depends critically on the specific interaction between matrix size and block size.

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	1200	0.01	ReLU	5.0	553.23	996.33	111.13	368.43	79.91	98.25
2-64-64-1	2000	0.01	Sigmoid	3.0	553.23	996.33	114.74	378.40	79.26	98.15
2-32-32-1	2000	0.01	Sigmoid	2.0	553.23	996.33	115.66	394.61	79.09	97.99
2-64-64-1	2000	0.01	ReLU	5.0	553.23	996.33	119.37	414.64	78.42	97.78
2-64-64-1	1500	0.01	ReLU	3.0	553.23	996.33	119.55	414.88	78.39	97.78

Tab. 4.19.: Performance of Feedforward Neural Networks with High Regularization and Extended Training

FFNN (Filtered Dataset)

Applying the expanded feature set (Nodes, MSIZE, BSIZE) to the filtered dataset yields the most accurate predictions in this study.

Table 4.20 shows a dramatic reduction in error. The MAE drops to as low as **82.50**, resulting in an improvement of over **82%** compared to the USL baseline. The results confirm that the primary bottleneck in previous tests was the loss of information

when aggregating matrix and block sizes into a single "Input Size" variable. When the neural network is trained on filtered data, it can virtually eliminate the prediction error, and capture the complex, non-linear scaling behavior of the system.

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	2000	0.01	ReLU	2.0	464.79	870.89	82.50	253.66	82.25	95.71
2-64-64-1	1200	0.01	ReLU	5.0	464.79	870.89	82.63	228.79	82.22	96.51
2-64-64-1	800	0.01	ReLU	10.0	464.79	870.89	83.18	191.86	82.10	97.55
2-64-64-1	1200	0.01	ReLU	3.0	464.79	870.89	83.72	205.88	81.99	97.17
2-16-16-1	500	0.1	ReLU	3.0	464.79	870.89	84.07	161.74	81.91	98.26

Tab. 4.20.: Performance of Feedforward Neural Networks with Strong Regularization and Low Hybrid Error

XGBoost (Original Dataset)

In Test 2, XGBoost leverages the separated input features (`MSIZE` and `BSIZE`) to achieve a substantial performance gain over the previous test.

Table 4.21 shows that the MAE drops to **127.96**, a **76.87%** improvement over the USL baseline. Compared to Test 1 (where MAE was ≈ 318), this confirms that decision trees benefit immensely from granular feature access, allowing them to create specific split rules for different block sizes.

While the error is slightly higher than the corresponding Neural Network result (MAE ≈ 111), the XGBoost model remains highly effective, even with a high asymmetric penalty ($\lambda = 100.0$) applied to control for the noise in the original dataset.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	6	200	50	100.0	553.23	996.33	127.96	406.02	76.87	96.84
mae	0.1	6	200	100	100.0	553.23	996.33	127.96	406.02	76.87	96.84
mae	0.1	6	200	150	100.0	553.23	996.33	127.96	406.02	76.87	96.84
mae	0.1	6	200	200	100.0	553.23	996.33	127.96	406.02	76.87	96.84
mae	0.1	6	200	500	100.0	553.23	996.33	127.96	406.02	76.87	96.84

Tab. 4.21.: Performance of XGBoost with Asymmetric Loss (Max Depth = 6, Penalty = 100.0, Boost Rounds = 200)

XGBoost (Filtered Dataset)

In the filtered dataset analysis for Test 2, XGBoost achieves its highest accuracy levels. The Mean Absolute Error (MAE) is reduced to **93.77**, marking a significant **79.83%** improvement over the USL baseline.

The model captures nearly all the variance in the execution time. The use of a moderate penalty ($\lambda = 50.0$) helped balance the model tendency to underestimate execution times.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
rmse	0.1	4	200	100	50.0	464.79	870.89	93.77	177.51	79.83	97.74
mae	0.1	4	200	50	50.0	464.79	870.89	95.02	179.96	79.56	97.68
mae	0.1	4	200	100	50.0	464.79	870.89	95.02	179.96	79.56	97.68
mae	0.1	4	200	150	50.0	464.79	870.89	95.02	179.96	79.56	97.68
mae	0.1	4	200	200	50.0	464.79	870.89	95.02	179.96	79.56	97.68

Tab. 4.22.: Performance of XGBoost with Asymmetric Loss (Max Depth = 4, Penalty = 50.0, Boost Rounds = 200)

4.11.6 Replacement of Input Size with the MSIZE in USL (Test 3)

In Test 3, we modified the calculation of the parameters for the Universal Scalability Law (USL) formula using `MSIZE` as the load parameter, under the hypothesis that matrix dimensions determine the distribution of parallel tasks. The neural network inputs remained the same as in Test 2 (Nodes, MSIZE, BSIZE).

Calculation of the theoretical model

By the calculation of the theoretical model, we obtain the results in Table 4.23 for the original dataset, and the results in Table 4.24 for the filtered dataset.

Parameter	Symbol	Value	Description
Serialization	α	-0.8855	Contention overhead
Coherency	β	0.1715	Crosstalk overhead
Baseline Time	T_1	67.3718	Single-node execution time (s)

Tab. 4.23.: Estimated USL Model Parameters for the Original Dataset

Parameter	Symbol	Value	Description
Serialization	α	-0.8614	Contention overhead
Coherency	β	0.1640	Crosstalk overhead
Baseline Time	T_1	67.3718	Single-node execution time (s)

Tab. 4.24.: Estimated USL Model Parameters for the Filtered Dataset

In the plot in Figure 4.13 for the original dataset and in Figure 4.14 for the filtered dataset. We can see what is the movement of the USL function based on the number of nodes used.

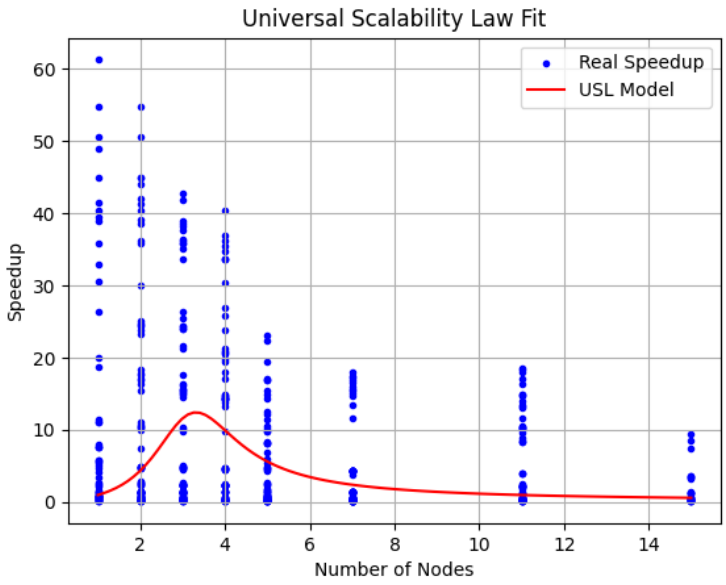


Fig. 4.13.: Plot of the USL function on the Original Dataset

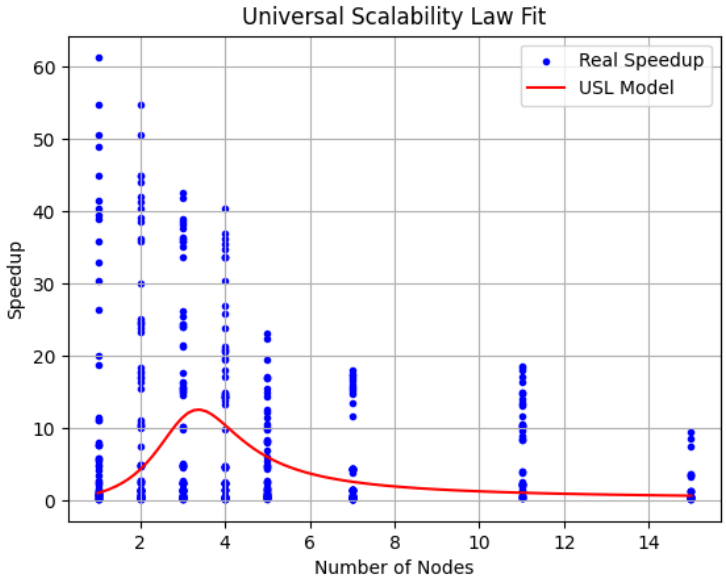


Fig. 4.14.: Plot of the USL function on the Filtered Dataset

FFNN (Original Dataset)

Table 4.25 highlights that relying only on `MSIZE` breaks the analytical model. The baseline USL error increase to **1324.51**, that is significantly higher than previous tests. This confirms that without accounting for Block Size, the USL cannot accurately model the system scalability.

Since the network receives BSIZE as an input, it learns the necessary corrections to "fix" the erroneous USL predictions. The model achieves an **88.82%** improvement, bringing the MAE down to ≈ 148. While this demonstrates the robustness of this hybrid model, the final error is higher than in Test 2 (≈ 111).

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-32-32-1	1500	0.1	Sigmoid	2.0	1324.51	2293.88	148.11	474.47	88.82	95.28
2-16-16-1	1500	0.1	Sigmoid	2.0	1324.51	2293.88	150.66	378.97	88.63	96.99
2-16-16-1	1200	0.1	Sigmoid	2.0	1324.51	2293.88	151.35	465.98	88.57	95.45
2-64-64-1	1500	0.01	ReLU	10.0	1324.51	2293.88	153.19	473.75	88.43	95.30
2-32-32-1	1200	0.01	Tanh	2.0	1324.51	2293.88	156.65	480.21	88.17	95.17

Tab. 4.25.: Performance of Feedforward Neural Networks on High-Variance USL Regime

FFNN (Filtered Dataset)

Table 4.26 shows that the USL baseline remains highly inaccurate (MAE ≈ 1204), proving that MSIZE alone is insufficient to predict performance, even when the data is clean. However, the neural network achieves a massive **91.80%** improvement, reducing the MAE to **98.84**.

This result is comparable to the best performance in Test 2. It indicates that when the analytical model (USL) fails catastrophically, the neural network effectively takes over, learning the correct relationships directly from the MSIZE and BSIZE inputs to deliver precise predictions ($R^2 \approx 97.8\%$).

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	1000	0.01	ReLU	5.0	1204.73	1883.72	98.84	238.76	91.80	97.78
2-64-64-1	600	0.01	ReLU	2.0	1204.73	1883.72	104.01	263.03	91.37	97.30
2-64-64-1	900	0.01	ReLU	3.0	1204.73	1883.72	104.84	258.29	91.30	97.40
2-32-32-1	1500	0.1	ReLU	2.0	1204.73	1883.72	105.43	229.95	91.25	97.94
2-64-64-1	1500	0.01	ReLU	3.0	1204.73	1883.72	107.03	296.55	91.12	96.57

Tab. 4.26.: Performance of Feedforward Neural Networks (High USL Error, Strong Hybrid Recovery)

XGBoost (Original Dataset)

In Test 3 with the original dataset, XGBoost demonstrates remarkable resilience. Despite the USL baseline producing a massive error (MAE ≈ 1102) due to the reliance only on `MSIZE`, XGBoost virtually eliminates this weakness.

Table 4.27 shows that the hybrid model achieves a Mean Absolute Error of just **86.73**. This is the single largest improvement percentage observed in any test configuration so far. The consistency of the results across different early stopping thresholds and depths confirms that the gradient boosting algorithm rapidly converges to a solution.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
rmse	0.1	4	200	50	10.0	1102.92	1945.28	86.73	213.50	92.14	92.50
mae	0.1	6	200	50	2.0	1102.92	1945.28	86.76	222.23	92.13	92.50
mae	0.1	6	200	100	2.0	1102.92	1945.28	86.76	222.23	92.13	92.50
mae	0.1	6	200	150	2.0	1102.92	1945.28	86.76	222.23	92.13	92.50
mae	0.1	6	200	200	2.0	1102.92	1945.28	86.76	222.23	92.13	92.50

Tab. 4.27.: Performance of XGBoost with Asymmetric Loss (High USL Error, Near-Perfect R^2 on Hybrid)

XGBoost (Filtered Dataset)

The results for XGBoost on the filtered dataset in Test 3 reveal an interesting divergence from previous patterns. Table 4.28 shows that while the model significantly improves upon the USL baseline (reducing MAE from ≈ 1204 to ≈ 266, an improvement of **77.92%**), it performs slightly worse compared to the Feedforward Neural Network on the same task (which achieved MAE ≈ 98).

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	6	200	50	3.0	1204.73	1883.72	266.04	528.45	77.92	92.71
mae	0.1	6	200	100	3.0	1204.73	1883.72	266.04	528.45	77.92	92.71
mae	0.1	6	200	150	3.0	1204.73	1883.72	266.04	528.45	77.92	92.71
mae	0.1	6	200	200	3.0	1204.73	1883.72	266.04	528.45	77.92	92.71
mae	0.1	6	200	500	3.0	1204.73	1883.72	266.04	528.45	77.92	92.71

Tab. 4.28.: Performance of XGBoost with Asymmetric Loss (Max Depth = 6, Penalty = 3.0, Boost Rounds = 200)

4.11.7 Improvement of the training using K-Fold Validation (Test 4)

In Test 4, we moved beyond simple train-test splits and implemented K-Fold Cross Validation. This technique ensures that every data point is used for both training and validation across different iterations, providing a much more rigorous assessment of the model generalization capability, especially on the noisy original dataset.

FFNN (Original Dataset)

The results in Table 4.29 reflect a more realistic performance estimate than previous tests. The average MAE of **224.88** is higher than the Test 2, but the consistency across folds, which confirms that the model is stable. Crucially, the hybrid model achieves an **84.96%** improvement over the USL baseline. The fact that the USL error is extremely high suggests that in certain folds, the analytical model fails completely (likely due to outliers).

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	200	0.01	ReLU	2.0	1518.72	2764.75	224.88	554.03	84.96	95.36
2-32-32-1	200	0.05	ReLU	2.0	1518.72	2764.75	228.51	557.19	84.66	95.41
2-64-64-1	200	0.01	ReLU	2.0	1518.72	2764.75	229.36	625.55	84.74	94.21
2-64-64-1	200	0.01	ReLU	3.0	1518.72	2764.75	230.34	562.20	84.70	95.34
2-64-64-1	150	0.01	ReLU	3.0	1518.72	2764.75	239.62	574.79	83.95	94.87

Tab. 4.29.: Feedforward Neural Network Performance on High-Variance USL Task

FFNN (Filtered Dataset)

Applying K-Fold Cross Validation to the filtered dataset confirms the high performance of the hybrid neural network. As shown in Table 4.30, the USL baseline produces a very high error (MAE ≈ 1204), consistent with previous findings that the physical model struggles when Block Size is ignored. However, the neural network achieves a massive improvement, reducing the MAE to just **98.84**. The consistency of this result with the single-split experiments (Test 3) validates the reliability of the model

Architecture	Epochs	LR	Activation	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
2-64-64-1	1000	0.01	ReLU	5.0	1204.73	1883.72	98.84	238.76	91.80	97.78
2-64-64-1	600	0.01	ReLU	2.0	1204.73	1883.72	104.01	263.03	91.37	97.30
2-64-64-1	900	0.01	ReLU	3.0	1204.73	1883.72	104.84	258.29	91.30	97.40
2-32-32-1	1500	0.1	ReLU	2.0	1204.73	1883.72	105.43	229.95	91.25	97.94
2-64-64-1	1500	0.01	ReLU	3.0	1204.73	1883.72	107.03	296.55	91.12	96.57

Tab. 4.30.: Feedforward Neural Network Performance: High USL Error with Strong Hybrid Recovery

XGBoost (Original Dataset)

The K-Fold Cross Validation results for XGBoost on the original dataset demonstrate the model is robust, even with some criticalities. Table 4.31 shows that the USL baseline yields a very high Mean Absolute Error (≈ 1518), confirming that `MSIZE` alone is a poor predictor of performance. XGBoost successfully corrects this, achieving an **81.38%** improvement and decreasing the MAE to **284.80**.

While this is a substantial recovery, the performance is slightly lower than that of the Feedforward Neural Network in the same K-Fold configuration. This suggests that in the presence of high noise (Original Dataset) and a weak physical baseline, the activation function of the neural network offers a slight advantage over the decision boundaries of XGBoost.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	4	200	50	10.0	1518.72	2764.75	284.80	778.49	81.38	91.64
rmse	0.1	4	200	50	10.0	1518.72	2764.75	284.86	779.23	81.38	91.62
mae	0.1	4	200	50	5.0	1518.72	2764.75	291.29	813.25	81.02	90.98
rmse	0.1	4	200	50	5.0	1518.72	2764.75	291.31	813.33	81.02	90.97
mae	0.1	6	200	50	10.0	1518.72	2764.75	296.49	861.29	80.87	89.91

Tab. 4.31.: XGBoost with Asymmetric Loss (5-Fold CV): High USL Error with Strong Hybrid Recovery

XGBoost (Filtered Dataset)

Table 4.32 indicates that XGBoost reduces the USL baseline error from **1170.87** to **165.29**, achieving an improvement of **85.56%**. However, similar to the findings in Test 3, XGBoost yields a higher residual error compared to the Feedforward Neural Network. This suggests that while XGBoost is excellent at capturing sharp decision boundaries, the Neural Network is slightly more efficient at approximating the correction functions.

Metric	Eta	Max Depth	Boost Rounds	Early Stop	Penalty	MAE USL	RMSE USL	MAE Hybrid	RMSE Hybrid	Improve MAE (%)	R^2 (%)
mae	0.1	4	200	50	3.0	1170.87	2000.70	165.29	491.61	85.56	92.70
mae	0.1	4	200	50	5.0	1170.87	2000.70	166.84	470.94	85.51	93.34
rmse	0.1	4	200	50	5.0	1170.87	2000.70	167.21	471.19	85.48	93.33
rmse	0.1	4	200	50	3.0	1170.87	2000.70	167.37	493.70	85.39	92.64
mae	0.1	4	200	50	2.0	1170.87	2000.70	168.13	498.82	85.34	92.53

Tab. 4.32.: XGBoost with Asymmetric Loss (5-Fold CV): Medium USL Error Regime

4.12 Evaluation of Models

This chapter presents the evaluation of the models showing the results obtained by each model. The first comparison shows the predicted execution time obtained by the model along every item of the dataset. Then, the chapter describes the result obtained by the complex models for the prediction of the residuals of the Universal Scalability Law on a training set and a testing set, useful to understand if the models suffer underfitting or overfitting. Lastly, we show the tests made on real data to understand and evaluate the performance with never-seen data.

Test	Description
Test 1	Inputs: Nodes and Input Size. Baseline: calculated via Input Size. Datasets: Original and filtered.
Test 2	Inputs: Nodes, MSIZE, BSIZE. Baseline: calculated via Input Size. Datasets: Original and filtered.
Test 3	Inputs: Nodes, MSIZE, BSIZE. Baseline: calculated via MSIZE (representing parallel tasks). Datasets: Original and filtered.
Test 4	Same setup as Test 3, utilizing k-fold cross-validation.

Tab. 4.33.: Summary of Experimental Tests

4.12.1 Discussion of Test 1 Results

Table 4.34 summarizes the percentage error metrics for the hybrid models trained using the simplified feature set with number of nodes and input size as inputs.

The analysis reveals three critical insights regarding this specific configuration:

- XGBoost achieves significantly lower average percentage errors ($\approx 24 - 29\%$) compared to the Feedforward Neural Network ($\approx 71 - 74\%$). This indicates that gradient boosting is far more robust when correcting a physical model based on aggregated input features.

- The FFNN models exhibit extreme variance and standard deviation ($\approx 96 - 113\%$), with maximum errors exceeding 1000%. This suggests that without granular access to `MSIZE` and `BSIZE`, the neural network fails to learn a stable correction function for the USL baseline.

- Overall, the error rates in Test 1 are significantly higher than those observed in the first stage (only machine learning model). This confirms that condensing workload parameters into a single "Input Size" variable is a loss of critical information.

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
XGBoost (Original)	24.31	16.51	53.02	604.22	24.58	0.04	197.40
XGBoost (Filtered)	29.09	19.66	21.06	1262.95	35.54	0.06	399.69
FFNN (Original)	74.41	41.18	51.82	9171.76	95.77	0.28	739.27
FFNN (Filtered)	71.35	33.60	47.04	12781.42	113.05	0.01	1066.58

Tab. 4.34.: Prediction Error Metrics – Test 1

4.12.2 Discussion of Test 2 Results

Table 4.35 presents the error metrics for the second test configuration, where the models were provided with explicit `MSIZE` and `BSIZE` inputs rather than an aggregated value.

The results highlight a dramatic improvement in predictive accuracy compared to Test 1:

- The best result in this entire study was achieved by **XGBoost on the Original Dataset**, with an exceptionally low average error of **6.24%** and a median error of just **1.38%**. This confirms that when the model has granular access to block size information, it can almost perfectly correct the theoretical USL baseline, even in the presence of noise.

- Across both datasets, XGBoost consistently yields lower variance and lower maximum errors than the neural network. The FFNN, particularly on the filtered dataset, shows significant instability (Max % Error $\approx 980\%$), suggesting that it struggles to converge on a stable correction function.

- Counter-intuitively, the percentage errors are higher on the filtered dataset than on the original one (e.g., XGBoost 14.11% vs 6.24%). This is likely a statistical artifact: filtering often removes large, stable execution times (outliers). The remaining dataset may contain smaller execution time values where even small absolute deviations result in large percentage errors.

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
XGBoost (Original)	6.24	1.38	0.21	170.57	13.06	0.00	152.12
XGBoost (Filtered)	14.11	6.41	0.79	396.97	19.92	0.00	147.69
FFNN (Original)	18.24	4.03	0.59	1668.40	40.85	0.01	469.40
FFNN (Filtered)	28.28	7.81	0.03	5742.78	75.78	0.01	980.50

Tab. 4.35.: Prediction Error Metrics – Test 2

4.12.3 Discussion of Test 3 Results

Table 4.36 displays the error metrics for Test 3, where the analytical USL model was deliberately weakened by using only `MSIZE` as the input load, while the machine learning models retained access to the full feature set.

- Despite the bad USL baseline, XGBoost maintained high accuracy, with the filtered dataset achieving an average error of **13.20%**. This is an improvement over the filtered result in Test 2 (14.11%), suggesting that XGBoost effectively ignores the erroneous analytical input and relies on the raw features to form its predictions.

- Unlike XGBoost, the Feedforward Neural Network suffered a noticeable drop in performance compared to Test 2. The average error on the original dataset rose from $\approx 18\%$ to $\approx 27\%$. This indicates that the neural network is more sensitive to the quality of the "physical" input; when the USL provides large errors, the network struggles more than decision trees to compensate.

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
XGBoost (Original)	15.62	5.45	0.83	642.66	25.35	0.01	266.50
XGBoost (Filtered)	13.20	3.70	1.78	976.77	31.25	0.01	526.27
FFNN (Original)	27.07	7.35	0.37	3084.36	55.54	0.00	548.98
FFNN (Filtered)	32.81	6.94	0.44	5515.88	74.27	0.00	917.11

Tab. 4.36.: Prediction Error Metrics – Test 3

4.12.4 Discussion of Test 4 Results (K-Fold Validation)

Table 4.37 presents the error metrics obtained using K-Fold Cross Validation. This test provides the most statistically robust assessment of model performance, as it evaluates predictive accuracy across the entire dataset rather than a single train-test split.

The results reinforce the superiority of the tree-based hybrid approach:

- XGBoost shows remarkable consistency, achieving an average error of **15.60%** on the original dataset and **16.03%** on the filtered dataset. The low variance and standard deviation ($\approx 21 - 22\%$) confirm that the model does not overfit to specific data subsets and provides reliable predictions even on unseen folds.

- The average error for XGBoost in Test 4 ($\approx 15.6\%$) is higher than the "best case" result seen in Test 2 ($\approx 6.2\%$). This is expected. The single-split testing (Test 2) can sometimes yield overly optimistic results if the test set happens to be easy. K-Fold validation reveals the true, generalized performance level of the model.

- The Feedforward Neural Network continues to struggle with relative accuracy, showing average errors of over **40%** and very high standard deviations ($\approx 87 - 99\%$). This high variability suggests that while the FFNN can approximate the function well in some folds, it fails to converge reliably in others, making XGBoost the safer choice.

Model	Avg % Error	Median % Error	Mode % Error	Variance % Error	Std Dev % Error	Min % Error	Max % Error
XGBoost (Original)	15.60	5.63	0.60	489.21	22.12	0.01	131.58
XGBoost (Filtered)	16.03	6.82	0.50	428.98	20.71	0.01	104.89
FFNN (Original)	44.07	9.55	1.91	7683.75	87.66	0.01	653.65
FFNN (Filtered)	42.60	10.11	2.34	9812.42	99.06	0.01	820.40

Tab. 4.37.: Prediction Error Metrics – Test 4

4.12.5 Best Results

Table 4.38 details the generalization capabilities of the models by comparing Training and Testing errors.

- The **XGBoost (Filtered)** model in Test 2 achieves the optimal balance. The minimal difference between Training MAE (108.59) and Test MAE (106.60) indicates the model successfully captured the underlying trend without memorizing noise.

- Across Test 2 and 3, the FFNN consistently exhibits overfitting, with training errors significantly lower than testing errors. However, under the robust K-Fold validation in Test 4, the **FFNN (Filtered)** emerges as a strong runner-up, balancing low error with reasonable generalization.

Conversely, the diagnosis reveals specific configurations that performed poorly:

- **Underfitting (Test 1):** Models trained on the original dataset with aggregated inputs consistently showed "Test $<$ Train" error. This underfitting indicates that the simplified input feature (aggregated size) was insufficient to capture the complexity of the training data.

- **Instability (Test 4):** While XGBoost excelled in Test 2, it failed to converge during K-Fold validation (Test 4), and it shows poor performance with high

errors. This suggests that XGBoost is less stable than the FFNN when correcting a deficient physical model across randomized folds.

Model	Variant	Train MAE	Test MAE	Train RMSE	Test RMSE	Diagnosis
				Test 1		
FFNN	Original	413.59	359.01	677.92	577.53	Underfitting (Test < Train)
FFNN	Filtered	317.75	384.23	513.83	601.35	Overfitting
XGBoost	Original	641.77	468.60	1342.83	871.20	Underfitting
XGBoost	Filtered	379.19	346.45	813.35	615.88	Underfitting
				Test 2		
FFNN	Original	73.28	128.98	134.78	405.56	Overfitting
FFNN	Filtered	92.04	118.33	163.55	262.65	Overfitting
XGBoost	Original	230.78	206.51	612.47	544.39	Underfitting
XGBoost	**Filtered**	**108.59**	**106.60**	**272.64**	**188.12**	**Optimal / Best Fit**
				Test 3		
FFNN	Original	105.79	172.77	173.60	426.26	Overfitting
FFNN	Filtered	93.36	151.36	148.20	339.70	Overfitting
XGBoost	Original	608.43	497.30	995.90	771.50	Underfitting
XGBoost	Filtered	308.81	348.31	533.31	656.84	Overfitting
				Test 4		
FFNN	Original	150.26	122.28	303.33	212.06	Good (Test ≈ Train)
FFNN	Filtered	78.16	107.34	137.82	180.10	Slight Overfitting
XGBoost	Original	886.89	919.41	1678.91	1811.31	Poor Performance
XGBoost	Filtered	590.95	634.06	1070.86	1104.55	Poor Performance

Tab. 4.38.: Performance Comparison (Test 1 - 4). The XGBoost test 2 Filtered model remains the best, followed closely by FFNN test 4 Filtered.

All the plots of the results of the model on original dataset are showed in Appendix G.2 and all the plots of the results of the model on filtered dataset are showed in G.3.

Figure 4.15 shows the best result of the XGBoost on Filtered Data obtained in Test 2.

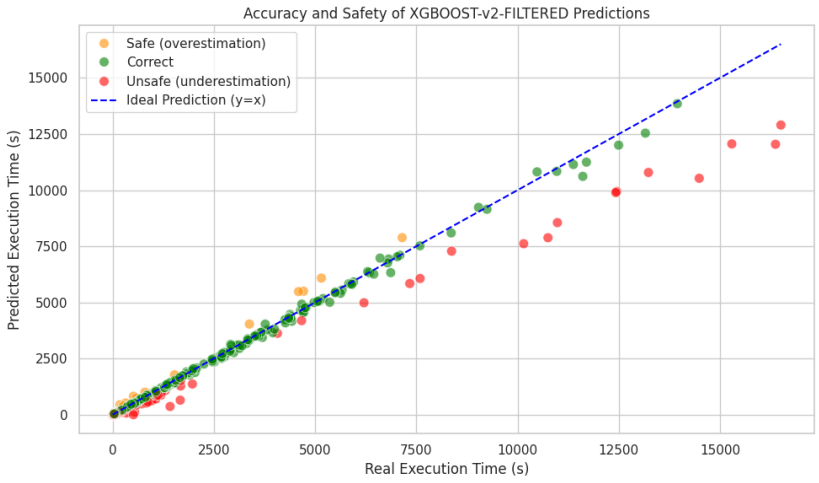


Fig. 4.15.: Plot of the prediction on the Filtered Dataset of the XGBoost

Figure 4.16 shows the best result of the XGBoost on Filtered Data residuals obtained in Test 2.

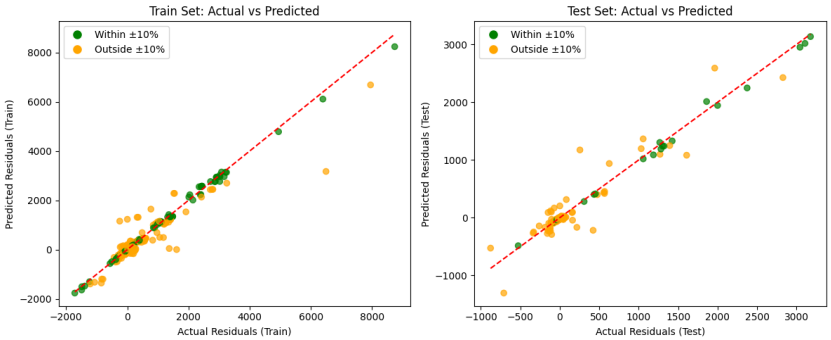


Fig. 4.16.: Plot of the prediction on the Filtered Dataset Residuals of the XGBoost

Figure 4.15 shows that the XGBoost model keeps strong linearity for execution times under 7500 seconds. While higher values show some divergence, the residual analysis in Figure 4.16 indicates overall reliability. Despite increased variance in the test set, some of residual predictions remain within the $\pm 10\%$ safety margins.

Figure 4.17 shows the best result of the FFNN on Filtered Data obtained in Test 2.

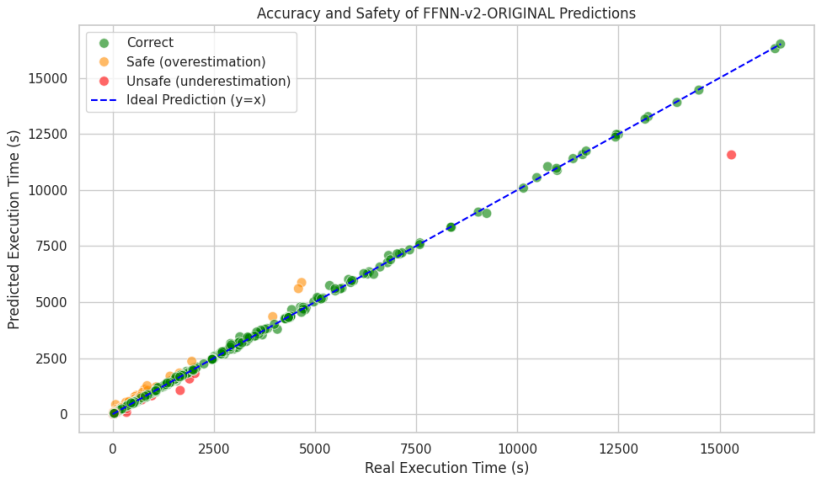


Fig. 4.17.: Plot of the prediction on the Filtered Dataset of the FFNN

Figure 4.18 shows the best result of the XGBoost on Filtered Data residuals obtained in Test 2.

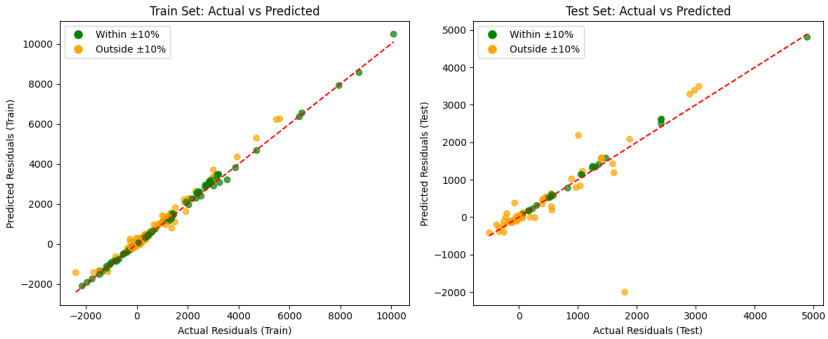


Fig. 4.18.: Plot of the prediction on the Filtered Dataset Residuals of the FFNN

Figure 4.17 shows good performance, with predictions tightly clustered around the ideal line and classified primarily as correct. This stability is reinforced by Figure 4.18, where the residual predictor shows a strong correlation across both datasets, keeping errors consistently within safety margins.

4.13 Model Accuracy Evaluation

To verify the practical applicability and generalization capability of the proposed solutions, we conducted a comprehensive validation test. In this phase, all developed models were evaluated on seven distinct input configurations (MSIZE, BSIZE, NUM_NODES). These specific configurations were excluded from the training dataset to ensure a rigorous evaluation of the model performance on unseen data. The selected points cover a diverse range of problem sizes and node counts, in order to test the ability of the model to interpolate within the known input space, thereby confirming that the models have learned the underlying patterns rather than simply memorizing the training examples.

The results referenced in Tables H.1 through H.7 allow for a direct cross-model comparison, leading to the observations described in the following subsections.

4.13.1 Performance on Large and Medium Workloads

For configurations resulting in substantial execution times, the hybrid models which use as inputs MSIZE, BSIZE, and NUM_NODES demonstrate exceptional accuracy.

- **The best models**: In scenarios such as 2 Nodes with BSIZE $= 600$ (Table H.3), the **XGBOOST-v2-ORIGINAL** and **FFNN-v3-FILTERED** models achieve errors as low as **1.64%** and **1.87%** respectively.

- **Good model**: Models trained on specific subsets (e.g., **FFNN-SIMPLE-SPECIFIC**) occasionally outperform general hybrids on medium workloads (e.g., Table H.2, Error 0.45%). However, these models often lack the flexibility to generalize outside their training envelope compared to the USL-guided hybrids.

4.13.2 Failure on Small Workloads

As shown in Table H.7, small execution times (e.g., 13 Nodes, ≈ 115s execution) present a challenge due to the dominance of system noise and communication overheads. In this regime, simple regression models struggle significantly, yielding errors exceeding 100%. However, complex models demonstrate better adaptability; while some configurations experience increased variance (e.g., XGBoost-v3), others like XGBOOST-SIMPLE maintain a much more useful accuracy (approx. 29% error) despite the volatile environment.

XGBoost v2 (Filtered)

XGBoost v2 proves to be the most consistent model for medium-to-large workloads. It maintains error rates below 12% in 5 out of 7 cases. The primary instability is observed in the highest node count configuration (13 Nodes), where the error increases significantly.

MSIZE	BSIZE	NODES	Actual	Predicted	Error %
9	500	2	212.9	189.00	11.23
22	257	2	310.4	319.00	2.77
22	600	2	4599.2	4937.00	7.34
43	345	5	1599.7	1464.00	8.48
37	200	6	187.4	207.00	10.46
70	200	8	775.0	923.00	19.10
35	200	13	114.8	230.00	100.35

Tab. 4.39.: Validation Results: XGBoost v2 (Filtered)

FFNN v3 (Filtered)

The FFNN v3 demonstrates good precision in specific configurations, achieving the lowest errors among all models for the largest workloads (1.87% and 1.39%). However, it exhibits higher variance than XGBoost v2, particularly in the 2-Node/500-Block configuration (44.20%) and the 13-Node configuration (38.15%), indicating that it is more sensitive to input scaling parameters.

MSIZE	BSIZE	NODES	Actual	Predicted	Error %
9	500	2	212.9	307.00	44.20
22	257	2	310.4	351.00	13.08
22	600	2	4599.2	4685.00	1.87
43	345	5	1599.7	1622.00	1.39
37	200	6	187.4	178.00	5.02
70	200	8	775.0	1426.00	84.00
35	200	13	114.8	71.00	38.15

Tab. 4.40.: Validation Results: FFNN v3 (Filtered)

FFNN test 4 (Filtered)

Despite using K-Fold cross-validation during training to improve generalization, FFNN test 4 performs less effectively on these specific validation points compared to test 3. While it handles the large workload (BSIZE $= 600$) well with 6.38% error, it suffers from severe overestimation in higher node counts (8 and 13 Nodes), indicating that the regularization applied in test 4 might have been too aggressive for these specific edge cases.

MSIZE	BSIZE	NODES	Actual	Predicted	Error %
9	500	2	212.9	265.00	24.47
22	257	2	310.4	386.00	24.36
22	600	2	4599.2	4306.00	6.38
43	345	5	1599.7	1547.00	3.29
37	200	6	187.4	231.00	23.27
70	200	8	775.0	2126.00	174.32
35	200	13	114.8	310.00	170.03

Tab. 4.41.: Validation Results: FFNN v4 (Filtered)

4.13.3 Summary of Best Results

While hybrid models offer a strong general solution, the validation phase highlights specific pure models that excel in distinct scenarios. Table **??** summarizes the optimal model for each specific validation case.

Superiority of Ensemble Methods: In general, tree-based ensemble methods (such as Random Forest and XGBoost) proved to be the most effective approach for this scenario. Unlike other architectures, these models naturally handle the tabular structure of system data and efficiently capture the non-linear "step" functions inherent in hardware scaling and resource allocation.

The detailed analysis confirms the hierarchy established during testing, identifying **XGBoost v2** and **FFNN v3** as the standout performers:

- **XGBoost v2** proves to be the most robust general-purpose model. It offers the best balance of reliability and accuracy, effectively capturing the non-linear scaling of communication overhead without overfitting to local noise.

- **FFNN v3** demonstrates superior performance on the largest workloads, achieving peak accuracy where complex interaction effects dominate. However, it remains less stable than the tree-based models in high-node settings, where outliers can skew the gradient descent.

Limitations on Low Execution Times

A critical limitation observed across all tested models is a significant performance degradation in scenarios with very low execution times. In these small-workload cases, system overhead, initialization latency, and operating system jitter dominate the actual computation time. This results in a low signal-to-noise ratio, making it statistically difficult for any model to distinguish the true execution pattern from system noise. Consequently, no model produced reliable predictions for the smallest input configurations.

All detailed results for every tested model and configuration are available in Appendix H.

5 Conclusion

5.1 Summary of Contributions

During the thesis, we worked on COMPSs and added new features, since we have the access directly to the source code. Hence, we contributed directly to the integration and development of new features such as the integration of the Profiler in COMPSs. We developed a robust, non-intrusive profiling system designed to operate across heterogeneous computing environments. We implemented an adaptive architecture with a three-tiered fallback mechanism, ensuring data collection works seamlessly on both modern systems (via `psutil` and `top`) and legacy HPC clusters (via custom `cgroup` readers). To promote interoperability, we standardized the output using the RO-Crate format, making the data about the Resource Usage adherent to FAIR principles. Eventually, we proved with extensive testing, that this solution introduces negligible overhead, consistently remaining below 5% to ensure it does not disrupt the scientific workflows it monitors. Moreover, we contributed to the open source Provenance Storage project. After having identified the limitations of its local Command Line Interface, we designed and implemented a REST API middleware using FastAPI. This contribution decoupled the interface from the database, enabling remote access and integration with automated external workflows.

Finally, we studied, developed and tested machine learning models for the prediction of the execution time of an application in an HPC environment. This research focused on rigorous methodology and real-world applicability rather than relying on synthetic data. For this reason, after the development and the integration of the profiler in the COMPSs project we needed to execute the tests in order to generate the dataset. After the data collection, we engineered a data pipeline to filter and clean real production data. We analyzed the data and removed the outliers to establish a stable ground truth for analysis. Initially, we implemented the simple models to define a baseline and then we implemented a hybrid modeling strategy that integrates the Universal Scalability Law (USL) with machine learning algorithms (XGBoost and Feedforward Neural Networks). While simple models frequently showed high variance, the residual learning approach successfully stabilized performance. Although the small workloads remain challenging to predict, the hybrid system significantly performed better than the pure machine learning models.

5.2 Future Work

The work presented in this thesis establishes a foundation for intelligent HPC workflow management, but there are several avenues for future research and development to fully maximize its potential.

Intelligent Resource Orchestration and SLURM Integration

The immediate next step is to integrate the predictive model directly with the Workload Manager (SLURM). Currently, users must estimate the wall-clock time for their jobs, which often leads to over-provisioning and inefficient queue usage.

- Automatic Time Assignment: The system could intercept job submissions and, based on the input parameters, automatically query the predictive model to assign the most suitable time limit. This would optimize cluster utilization by reducing the gap between requested and actual execution time.

- Configuration Recommender System: We propose developing an interactive interface where users describe their constraints (e.g., "fastest execution" vs. "lowest resource cost"). The model would then analyze the application inputs and recommend the optimal configuration (number of nodes, block size, and threads) to achieve the user specific goal.

Machine Learning models Enhancement

The current model relies on a static dataset trained on historical executions. To maintain accuracy over time as hardware ages or configurations change, the system should evolve into an Online Learning framework.

- Automated Retraining: Every new execution in the COMPSs workflow should automatically become a new data point. A pipeline could be established to verify this new ground truth and trigger a model retraining process periodically.

- Transfer Learning: Currently, the model is tuned for specific algebraic operations. Future work should investigate Transfer Learning techniques to adapt the weights of this model to different types of COMPSs applications without starting the training process from scratch.

Profiling Capabilities Enhancement

While the current profiler captures CPU, Memory, and I/O, modern HPC environments are increasingly heterogeneous.

- GPU and Accelerator Profiling: Extending the profiler to capture GPU metrics (usage, memory, temperature) is essential for supporting AI implementations and simulation workflows.

- Energy Efficiency (Green HPC): Integrating energy consumption metrics into the RO-Crate metadata would allow the predictive model to optimize not just for time, but for energy efficiency,.

Provenance Storage Enhancement

The ProvStor API currently serves as a backend middleware. Developing a frontend dashboard would significantly improve usability. This interface would allow system administrators to visualize the "health" of the workflows in real-time, view the specific hardware bottlenecks identified by the profiler, and visually compare the predicted execution time against the actual results.

Bibliography

[1] Gene M Amdahl. „Validity of the single processor approach to achieving large scale computing capabilities“. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485 (cit. on p. 83).

[2] Apache Software Foundation. *Apache Jena Fuseki Documentation*. Accessed: 2026-01-19. 2026 (cit. on p. 43).

[3] Maya Bechler-Speicher, Ido Amos, Ran Gilad-Bachrach, and Amir Globerson. „Graph neural networks use graphs when they shouldn't“. In: *Forty-first International Conference on Machine Learning*. 2024 (cit. on p. 69).

[4] Michael W Browne. „Cross-validation methods“. In: *Journal of mathematical psychology* 44.1 (2000), pp. 108–132 (cit. on p. 63).

[5] A. M. Chirkin et al. „Execution time estimation for workflow scheduling“. In: *Future Generation Computer Systems* (2017) (cit. on p. 68).

[6] Artem M Chirkin, Adam SZ Belloum, Sergey V Kovalchuk, et al. „Execution time estimation for workflow scheduling“. In: *Future generation computer systems* 75 (2017), pp. 376–387 (cit. on p. 55).

[7] RO-Crate Community. *RO-Crate. A Lightweight Approach to Research Object Packaging*. Specification portal. Research Object. Dec. 15, 2024. URL: `https://www.researchobject.org/ro-crate/` (visited on July 31, 2025) (cit. on pp. 4, 13, 29, 30).

[8] M. V. Devarakonda and R. K. Iyer. „Predictability of Process Resource Usage: A Measurement-Based Study of UNIX“. In: *IEEE Transactions on Software Engineering* 15.12 (1989) (cit. on pp. 4, 68).

[9] Paul Groth, Helena Cousijn, Tim Clark, and Carole Goble. „FAIR Data Reuse — the Path Through Data Citation“. In: *Data Intelligence* 2.1-2 (2020), pp. 78–86 (cit. on p. 1).

[10] Neil J Gunther. *Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services*. Springer, 2007 (cit. on p. 83).

[11] Jian Guo, Akihiro Nomura, Ryan Barton, Haoyu Zhang, and Satoshi Matsuoka. „Machine learning predictions for underestimation of job runtime on HPC system“. In: *Asian Conference on Supercomputing Frontiers*. Springer International Publishing Cham. 2018, pp. 179–198 (cit. on pp. 54, 68).

[12] X. Hu, L. Chu, J. Pei, W. Liu, and J. Bian. „Model complexity of deep learning: a survey“. In: *Knowledge and Information Systems* (2021) (cit. on p. 66).

[13]D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz, and B. M. Wilamowski. „Selection of Proper Neural Network Sizes and Architectures—A Comparative Study“. In: *IEEE Transactions on Industrial Informatics* 8.2 (2012), pp. 226–240 (cit. on p. 67).

[14]Naveen Kumar. *How Many Companies Use AI in 2025? (Global Data)*. Accessed: October 10, 2025. DemandSage. Apr. 2025. URL: https://www.demandsage.com/companies-using-ai/ (cit. on p. 1).

[15]Simone Leo, Michael R Crusoe, Laura Rodríguez-Navas, et al. „Recording provenance of workflow runs with RO-Crate“. In: *PLoS one* 19.9 (2024), e0309210 (cit. on p. 4).

[16]Francesc Lordan, Enric Tejedor, Jorge Ejarque, et al. „Servicess: An interoperable programming framework for the cloud“. In: *Journal of grid computing* 12.1 (2014), pp. 67–91 (cit. on p. 7).

[17]MinIO, Inc. *MinIO*. Accessed: 2026-01-19. 2026 (cit. on p. 44).

[18]Margaret Mitchell, Simone Wu, Andrew Zaldivar, et al. „Model Cards for Model Reporting“. In: *Proceedings of the Conference on Fairness, Accountability, and Transparency*. ACM, 2019, pp. 220–229 (cit. on p. 1).

[19]Luc Moreau and Paul Groth. „Provenance: An Introduction to PROV“. In: *Synthesis Lectures on the Semantic Web: Theory and Technology* 7.1 (2018), pp. 1–129 (cit. on p. 1).

[20]Adam Paszke, Sam Gross, Francisco Massa, et al. „Pytorch: An imperative style, high-performance deep learning library“. In: *Advances in neural information processing systems* 32 (2019) (cit. on pp. 4, 64).

[21]Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. „Scikit-learn: Machine learning in Python“. In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830 (cit. on pp. 4, 64).

[22]Suja Ramachandran, ML Jayalal, M Vasudevan, Sourish Das, and R Jehadeesan. „Combining machine learning techniques and genetic algorithm for predicting run times of high performance computing jobs“. In: *Applied Soft Computing* 165 (2024), p. 112053 (cit. on pp. 3, 63, 73).

[23]Raül Sirvent, Javier Conejero, Francesc Lordan, et al. „Automatic, Efficient and Scalable Provenance Registration for FAIR HPC Workflows“. In: *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)* (2022) (cit. on pp. 5, 30).

[24]W. Smith, I. Foster, and V. Taylor. *Predicting Application Run Times Using Historical Information*. Tech. rep. Argonne National Laboratory, Mathematics and Computer Science Division, 1998 (cit. on pp. 4, 73).

[25]M. Tanash, D. Andresen, H. Yang, and W. Hsu. „Ensemble Prediction of Job Resources to Improve System Performance for Slurm-Based HPC Systems“. In: *Practice and Experience in Advanced Research Computing (PEARC '21)*. 2021 (cit. on pp. 4, 63, 66).

[26]Enric Tejedor, Yolanda Becerra, Guillem Alomar, et al. „PyCOMPSs: Parallel computational workflows in Python“. In: *The International Journal of High Performance Computing Applications* 31.1 (2017), pp. 66–82 (cit. on pp. 5, 7).

[27]Tzu-Tsung Wong and Po-Yang Yeh. „Reliable accuracy estimates from k-fold cross validation“. In: *IEEE Transactions on Knowledge and Data Engineering* 32.8 (2019), pp. 1586–1594 (cit. on p. 63).

List of Figures

List of Tables

Listings

A

Example of *ro-crate-metadata.json* file


```
{
    "@id": "#overall.matmul_files.py.executionTime",
    "@type": "PropertyValue",
    "name": "executionTime",
    "propertyID": "https://w3id.org/ro/terms/compss#executionTime  ",
    "unitCode": "https://qudt.org/vocab/unit/MilliSEC  ",
    "value": "727124"
},
{
    "@id": "#gs10r2b26.cpuAvg",
    "@type": "PropertyValue",
    "name": "cpuAvg",
    "propertyID": "https://w3id.org/ro/terms/compss#cpuAvg  ",
    "unitCode": "https://qudt.org/vocab/unit/PERCENT  ",
    "value": "50.6"
},
{
    "@id": "#gs10r2b26.cpuMax",
    "@type": "PropertyValue",
    "name": "cpuMax",
    "propertyID": "https://w3id.org/ro/terms/compss#cpuMax  ",
    "unitCode": "https://qudt.org/vocab/unit/PERCENT  ",
    "value": "100.0"
},
{
    "@id": "#gs10r2b26.memAvg",
    "@type": "PropertyValue",
    "name": "memAvg",
    "propertyID": "https://w3id.org/ro/terms/compss#memAvg  ",
    "unitCode": "https://qudt.org/vocab/unit/PERCENT  ",
    "value": "15.82"
},
{
```

```
        "@id": "#gs10r2b26.memMin",
        "@type": "PropertyValue",
        "name": "memMin",
        "propertyID": "https://w3id.org/ro/terms/compss#memMin  ",
        "unitCode": "https://qudt.org/vocab/unit/PERCENT  ",
        "value": "13.21"
    },
    {
        "@id": "#gs10r2b26.memMax",
        "@type": "PropertyValue",
        "name": "memMax",
        "propertyID": "https://w3id.org/ro/terms/compss#memMax  ",
        "unitCode": "https://qudt.org/vocab/unit/PERCENT  ",
        "value": "16.68"
    },
```

B Matrix Multiplication Code

```
1  from pycompss.api.task import task
2  from pycompss.api.parameter import *
3
4
5  def load_block(fi):
6      b = []
7      f = open(fi, 'r')
8      for line in f:
9          split_line = line.split(' ')
10         r = []
11         for num in split_line:
12             r.append(float(num))
13         b.append(r)
14     f.close()
15     return b
16
17
18 def store_block(b, fi, size):
19     f = open(fi, 'w')
20     for row in b:
21         for j in range(size):
22             num = row[j]
23             f.write(str(num))
24             if j < size - 1:
25                 f.write(' ')
26         f.write('\n')
27
28
29 # ## TASK SELECTION ## #
30
31 @task(fa=FILE, fb=FILE, fc=FILE_INOUT)
32 def multiply(fa, fb, fc, size):
33     a = load_block(fa)
34     b = load_block(fb)
35     c = load_block(fc)
36     for i in range(size):
37         for j in range(size):
38             for k in range(size):
39                 c[i][j] += a[i][k] * b[k][j]
40     store_block(c, fc, size)
41
```

Listing B.1: Matrix Multiplication tasks execution

```
def initialize_variables():
    for i in range(MSIZE):
        A.append([])
        B.append([])
        C.append([])
        for j in range(MSIZE):
            A[i].append("A.%d.%d" % (i, j))
            B[i].append("B.%d.%d" % (i, j))
            C[i].append("C.%d.%d" % (i, j))


def fill_matrices():
    for c in ['A', 'B', 'C']:
        for i in range(MSIZE):
            for j in range(MSIZE):
                tmp = "%s.%d.%d" % (c, i, j)
                f = open(tmp, 'w')
                for _ in range(BSIZE):
                    for jj in range(BSIZE):
                        if c == 'C':
                            f.write('0.0')
                        else:
                            f.write('2.0')
                        if jj < BSIZE - 1:
                            f.write(' ')
                    f.write('\n')
                f.close()


# ## MAIN PROGRAM ## #

if __name__ == "__main__":
    import sys
    from matmul_tasks import multiply

    args = sys.argv[1:]
    MSIZE = int(args[0])
    BSIZE = int(args[1])

    A = []
    B = []
    C = []

    initialize_variables()
    fill_matrices()

    for i in range(MSIZE):
        for j in range(MSIZE):
            for k in range(MSIZE):
                multiply(A[i][k], B[k][j], C[i][j], BSIZE)

```

Listing B.2: Matrix Multiplication file initialization

C Pearson Correlation Analysis

C.1 Statistical Analysis of Original Dataset

Variable Pair	Coefficient	p-value	Is Significant?
MSIZE – EXEC_TIME	0.4343	3.73e-26	✓
MSIZE – CPU_AVG	0.7093	8.01e-83	✓
MSIZE – MEM_AVG	0.2206	2.60e-07	✓
BSIZE – EXEC_TIME	0.6155	2.09e-57	✓
BSIZE – CPU_AVG	0.3083	3.19e-13	✓
BSIZE – MEM_AVG	0.2877	1.23e-11	✓
NUM_NODES – EXEC_TIME	0.1488	0.0005	✓
NUM_NODES – CPU_AVG	0.4122	2.58e-23	✓
NUM_NODES – MEM_AVG	0.9856	0.00e+00	✓
NUM_CPUS – EXEC_TIME	0.1488	0.0005	✓
NUM_CPUS – CPU_AVG	0.4122	2.58e-23	✓
NUM_CPUS – MEM_AVG	0.9856	0.00e+00	✓
EXEC_TIME – CPU_AVG	0.5340	1.05e-40	✓
EXEC_TIME – MEM_AVG	0.2727	1.46e-10	✓
CPU_AVG – MEM_AVG	0.5332	1.45e-40	✓

Tab. C.1.: Pearson Correlation Results for CPU, memory, and execution time of original data

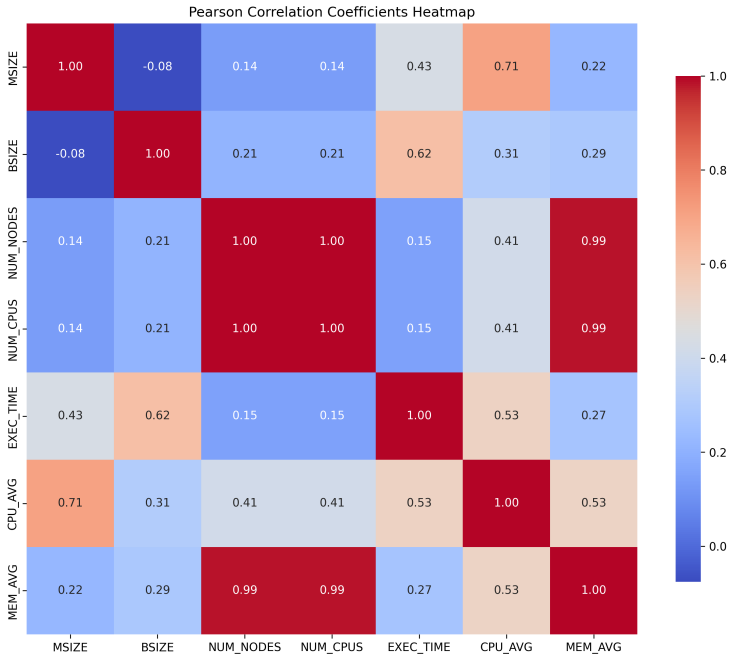


Fig. C.1.: Pearson distribution of the original dataset.

C.2 Statistical Analysis of Filtered Dataset

Variable Pair	Coefficient	p-value	Is Significant?
MSIZE – EXEC_TIME	0.2275	2.02e-07	✓
MSIZE – CPU_AVG	0.6400	9.06e-60	✓
MSIZE – MEM_AVG	0.0956	0.0313	✗
BSIZE – EXEC_TIME	0.6796	1.73e-70	✓
BSIZE – CPU_AVG	0.2428	3.10e-08	✓
BSIZE – MEM_AVG	0.2293	1.78e-07	✓
NUM_NODES – EXEC_TIME	0.0180	0.6853	✗
NUM_NODES – CPU_AVG	0.3371	6.14e-15	✓
NUM_NODES – MEM_AVG	0.9930	0.00e+00	✓
NUM_CPUS – EXEC_TIME	0.0180	0.6853	✗
NUM_CPUS – CPU_AVG	0.3371	6.14e-15	✓
NUM_CPUS – MEM_AVG	0.9930	0.00e+00	✓
EXEC_TIME – CPU_AVG	0.2701	6.31e-10	✓
EXEC_TIME – MEM_AVG	0.0917	0.0389	✗
CPU_AVG – MEM_AVG	0.4190	5.67e-23	✓

Tab. C.2.: Pearson Correlation Results for CPU, Memory, and Execution Time of filtered data

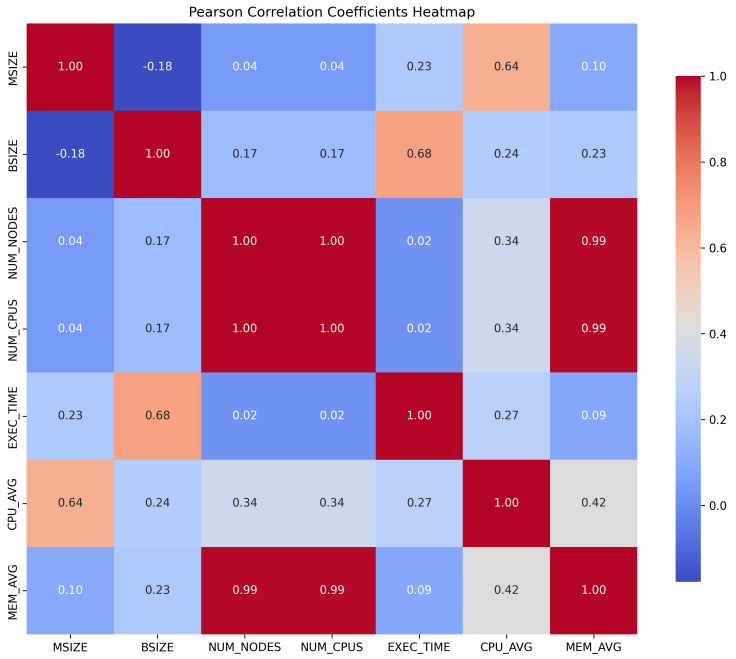


Fig. C.2.: Pearson distribution of the filtered dataset.

Spearman Correlation Analysis

D

D.1 Statistical Analysis of Original Dataset

Variable Pair	Coefficient	p-value	Is Significant?
MSIZE – EXEC_TIME	0.3215	2.09e-14	✓
MSIZE – CPU_AVG	0.6556	6.67e-67	✓
MSIZE – MEM_AVG	0.1469	0.0007	✓
BSIZE – EXEC_TIME	0.8715	5.47e-168	✓
BSIZE – CPU_AVG	0.5083	1.99e-36	✓
BSIZE – MEM_AVG	0.3154	8.51e-14	✓
NUM_NODES – EXEC_TIME	0.2224	1.88e-07	✓
NUM_NODES – CPU_AVG	0.3752	2.72e-19	✓
NUM_NODES – MEM_AVG	0.9882	0.00e+00	✓
NUM_CPUS – EXEC_TIME	0.2224	1.88e-07	✓
NUM_CPUS – CPU_AVG	0.3752	2.72e-19	✓
NUM_CPUS – MEM_AVG	0.9882	0.00e+00	✓
EXEC_TIME – CPU_AVG	0.6379	2.35e-62	✓
EXEC_TIME – MEM_AVG	0.3473	1.39e-16	✓
CPU_AVG – MEM_AVG	0.4684	1.79e-30	✓

Tab. D.1.: Spearman Correlation Results for CPU, Memory, and Execution Time of original data

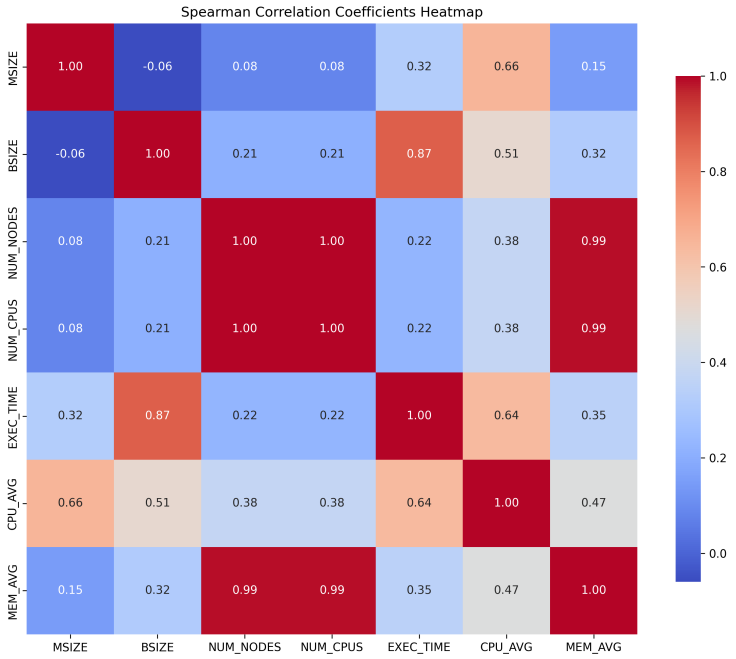


Fig. D.1.: Spearman distribution of the original dataset.

D.2 Statistical Analysis of Filtered Dataset

Variable Pair	Coefficient	p-value	Is Significant?
MSIZE – EXEC_TIME	0.2275	2.02e-07	✓
MSIZE – CPU_AVG	0.6400	9.06e-60	✓
MSIZE – MEM_AVG	0.0956	0.0313	✗
BSIZE – EXEC_TIME	0.6796	1.73e-70	✓
BSIZE – CPU_AVG	0.2428	3.10e-08	✓
BSIZE – MEM_AVG	0.2293	1.78e-07	✓
NUM_NODES – EXEC_TIME	0.0180	0.6853	✗
NUM_NODES – CPU_AVG	0.3371	6.14e-15	✓
NUM_NODES – MEM_AVG	0.9930	0.00e+00	✓
NUM_CPUS – EXEC_TIME	0.0180	0.6853	✗
NUM_CPUS – CPU_AVG	0.3371	6.14e-15	✓
NUM_CPUS – MEM_AVG	0.9930	0.00e+00	✓
EXEC_TIME – CPU_AVG	0.2701	6.31e-10	✓
EXEC_TIME – MEM_AVG	0.0917	0.0389	✗
CPU_AVG – MEM_AVG	0.4190	5.67e-23	✓

Tab. D.2.: Spearman Correlation Results for CPU, Memory, and Execution Time of filtered data

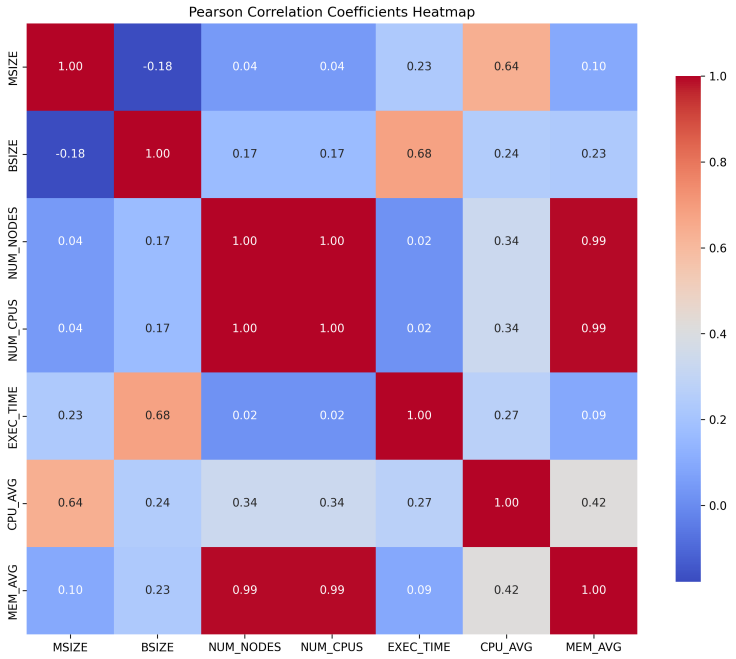


Fig. D.2.: Spearman distribution of the filtered dataset.

Principal Component Analysis

E

E.1 PCA of Original Dataset

Component	Variance Explained (%)
PC1	52.19
PC2	24.39
PC3	17.06
PC4	4.26
PC5	2.06

Tab. E.1.: Explained Variance Ratio (Original Dataset)

Variable	PC1	PC2	PC3	PC4	PC5
MSIZE	0.231	0.442	-0.586	0.129	0.624
BSIZE	0.227	0.272	0.716	-0.372	0.473
NUM_NODES	0.463	-0.352	-0.009	0.101	0.070
NUM_CPUS	0.463	-0.352	-0.009	0.101	0.070
EXEC_TIME	0.271	0.535	0.283	0.661	-0.347
CPU_AVG	0.386	0.366	-0.252	-0.621	-0.503
MEM_AVG	0.493	-0.251	0.004	0.043	-0.057

Tab. E.2.: Principal Component Loadings (Original Dataset)

E.2 PCA of Filtered Dataset

Component	Variance Explained (%)
PC1	47.28
PC2	24.77
PC3	20.48
PC4	5.64
PC5	1.78

Tab. E.3.: Explained Variance Ratio (Filtered Dataset)

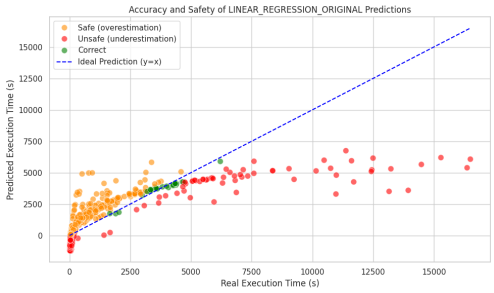
Variable	PC1	PC2	PC3	PC4	PC5
MSIZE	0.131	0.327	0.686	-0.343	0.536
BSIZE	0.192	0.471	-0.528	0.363	0.574
NUM_NODES	0.519	-0.237	-0.059	-0.115	0.026
NUM_CPUS	0.519	-0.237	-0.059	-0.115	0.026
EXEC_TIME	0.137	0.633	-0.257	-0.560	-0.448
CPU_AVG	0.321	0.361	0.418	0.638	-0.424
MEM_AVG	0.534	-0.167	-0.049	-0.062	-0.018

Tab. E.4.: Principal Component Loadings (Filtered Dataset)

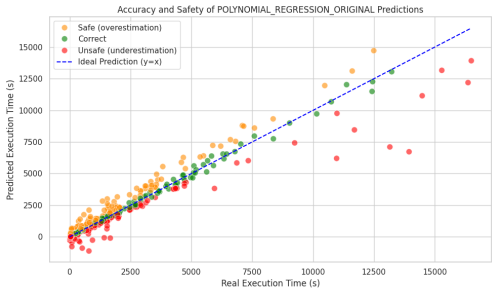
First Attempt Simple Model Plots

F

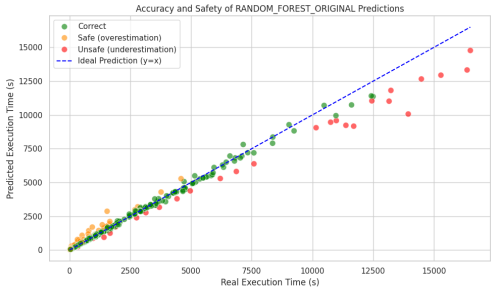
F.1 Plots of Original Dataset



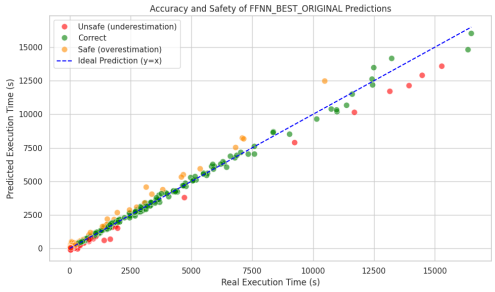
(a) Linear Regression on Original Dataset



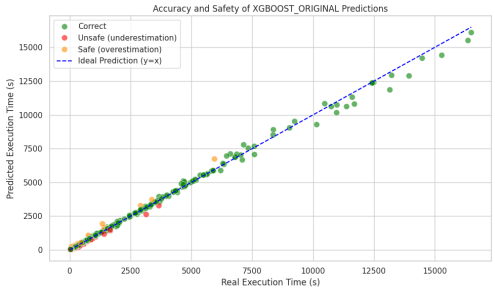
(b) Polynomial Regression on Original Dataset



(c) Random Forest on Original Dataset



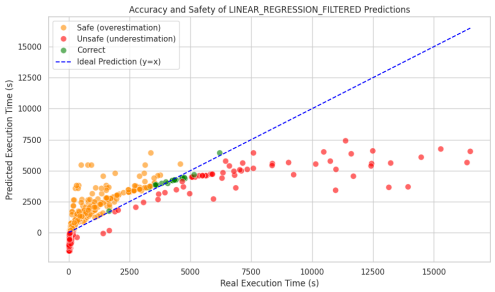
(d) Feedforward Neural Network on Original Dataset



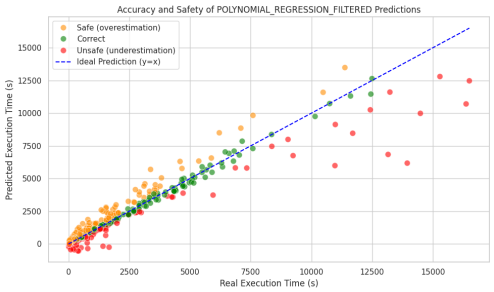
(e) XGBoost on Original Dataset

Fig. F.1.: Predictions of various simple models on the original dataset.

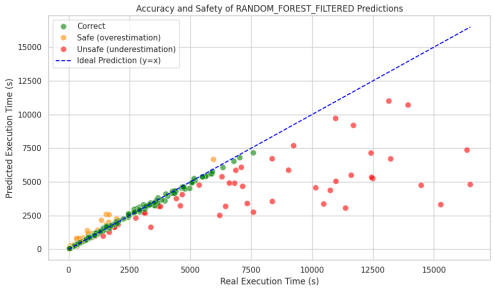
F.2 Plots of Filtered Dataset



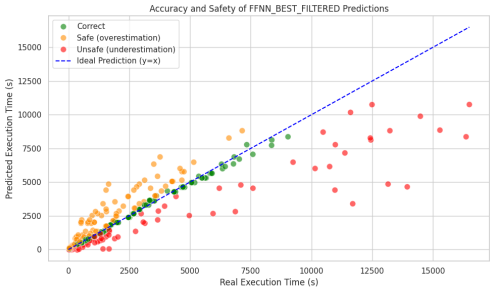
(a) Linear Regression on Filtered Dataset



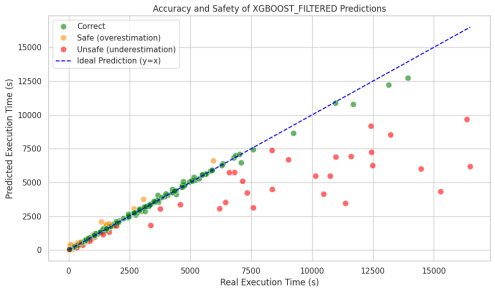
(b) Polynomial Regression on Filtered Dataset



(c) Random Forest on Filtered Dataset



(d) Feedforward Neural Network on Filtered Dataset



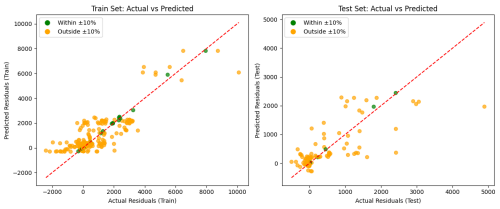
(e) XGBoost on Filtered Dataset

Fig. F.2.: Predictions of various simple models on the filtered dataset.

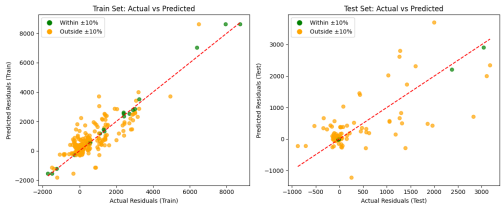
Plots of the Improved Models

G

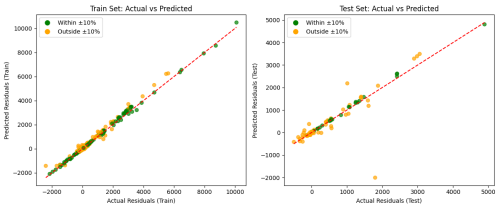
G.1 Plots of Residuals Prediction



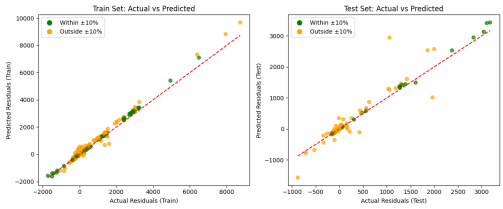
(a) FFNN test 1 – Original Dataset



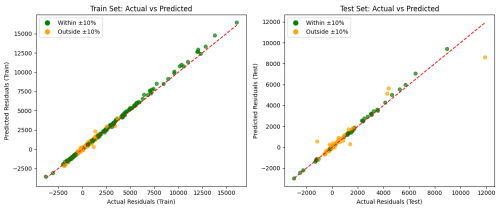
(b) FFNN test 1 – Filtered Dataset



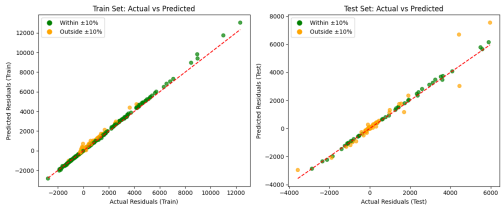
(c) FFNN v2 – Original Dataset



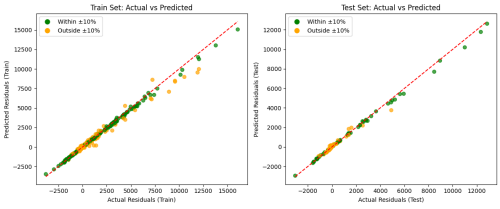
(d) FFNN test 2 – Filtered Dataset



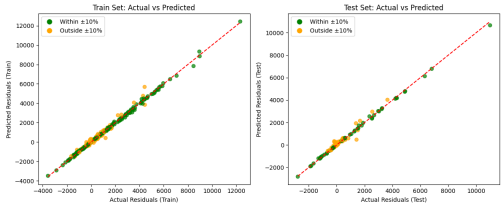
(e) FFNN test 3 – Original Dataset



(f) FFNN test 3 – Filtered Dataset

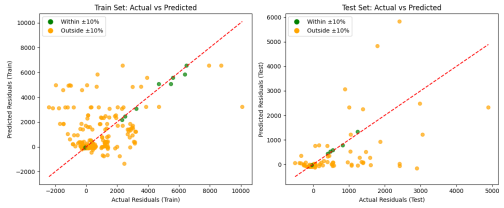


(g) FFNN test 4 – Original Dataset

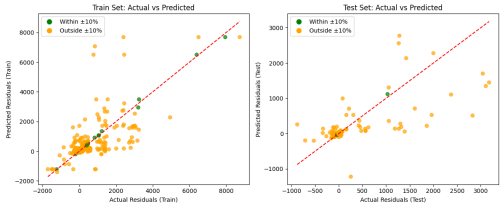


(h) FFNN test 4 – Filtered Dataset

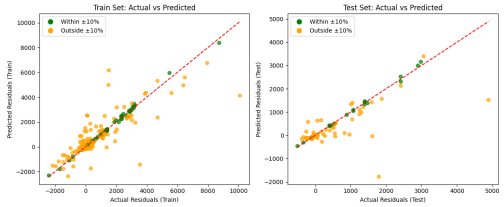
Fig. G.1.: USL residual plots for Feedforward Neural Network (FFNN) variants tests 1-4 on original and filtered datasets.



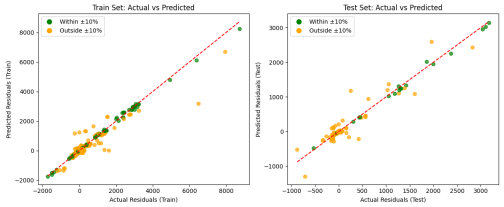
(a) XGBoost v1 – Original Dataset



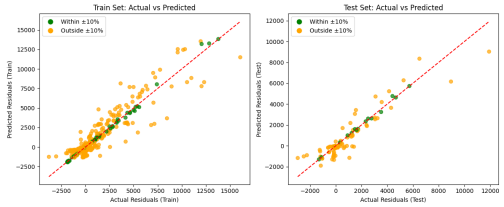
(b) XGBoost test 1 – Filtered Dataset



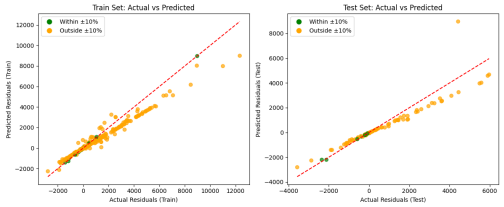
(c) XGBoost test 2 – Original Dataset



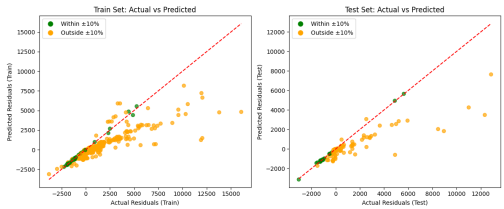
(d) XGBoost test 2 – Filtered Dataset



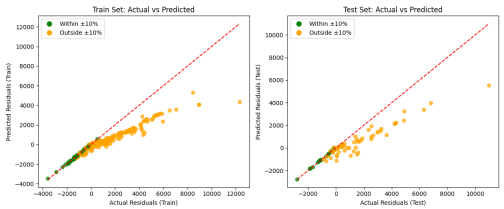
(e) XGBoost test 3 – Original Dataset



(f) XGBoost test 3 – Filtered Dataset



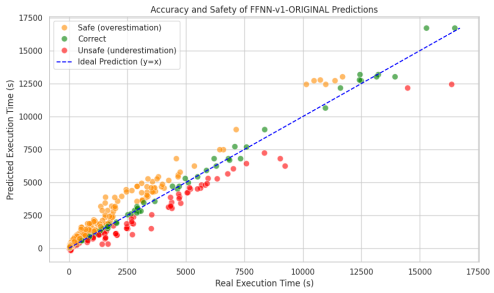
(g) XGBoost test 4 – Original Dataset



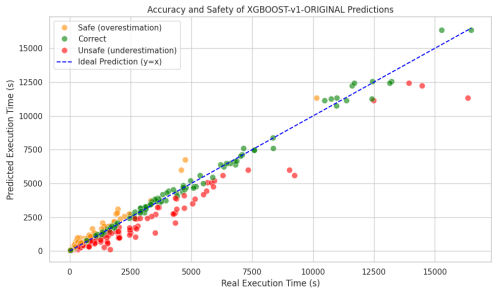
(h) XGBoost test 4 – Filtered Dataset

Fig. G.2.: USL residual plots for XGBoost variants v1–v4 on original and filtered datasets.

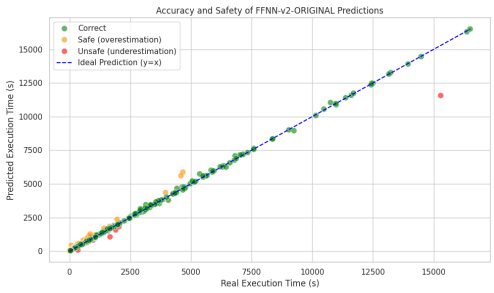
G.2 Plots of Original Dataset



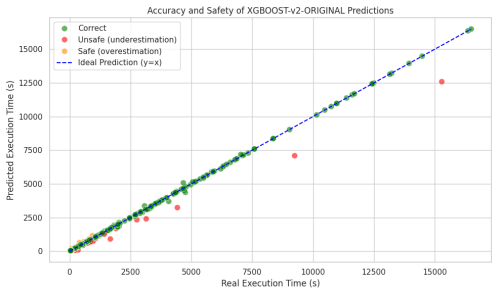
(a) FFNN test 1 – Original Dataset



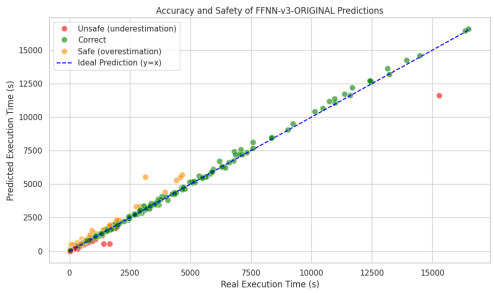
(b) XGBoost test 1 – Original Dataset



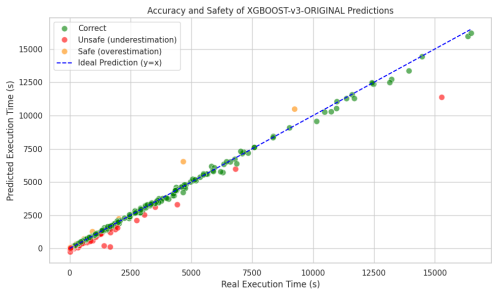
(c) FFNN test 2 – Original Dataset



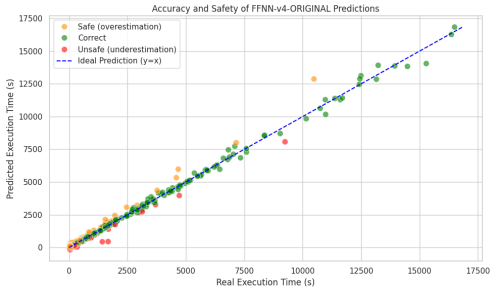
(d) XGBoost test 2 – Original Dataset



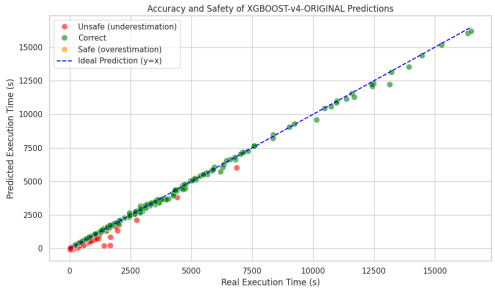
(e) FFNN test 3 – Original Dataset



(f) XGBoost test 3 – Original Dataset



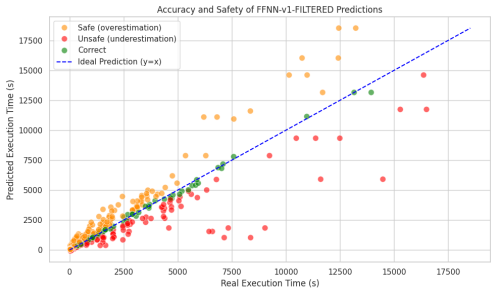
(g) FFNN test 4 – Original Dataset



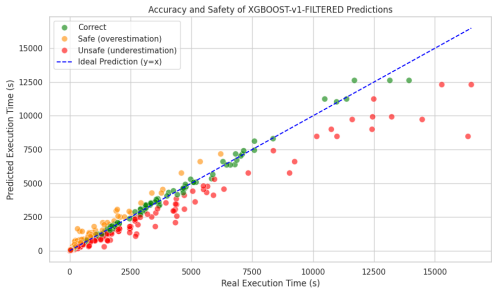
(h) XGBoost test 4 – Original Dataset

Fig. G.3.: Predictions of complex models FFNN and XGBoost on the original dataset.

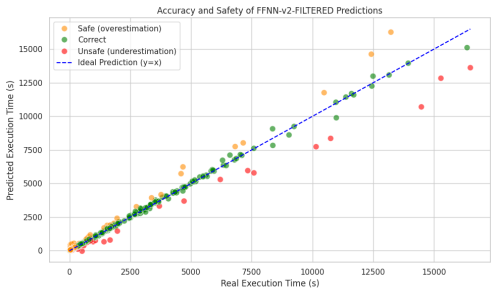
G.3 Plots of Filtered Dataset



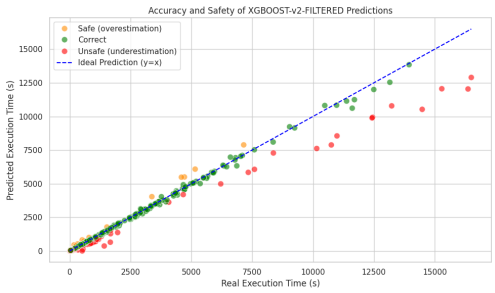
(a) FFNN test 1 – Filtered Dataset



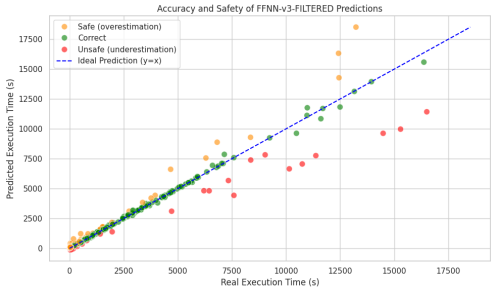
(b) XGBoost test 1 – Filtered Dataset



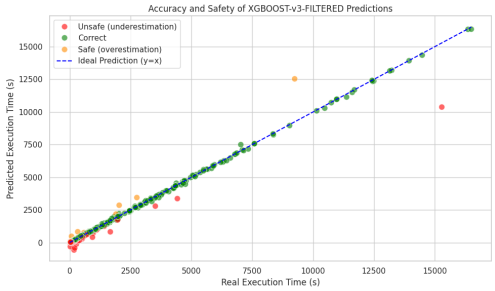
(c) FFNN test 2 – Filtered Dataset



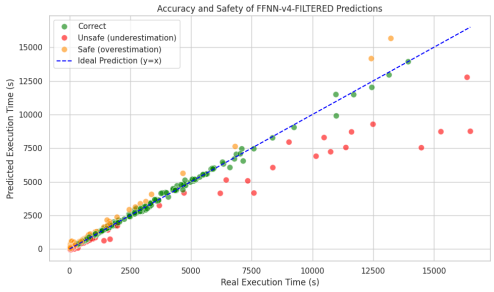
(d) XGBoost test 2 – Filtered Dataset



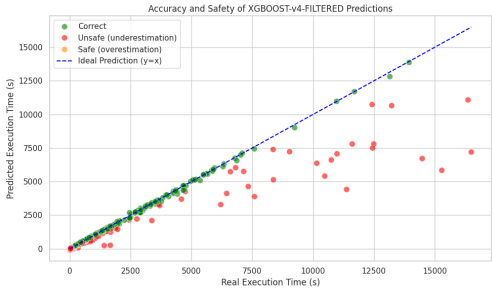
(e) FFNN test 3 – Filtered Dataset



(f) XGBoost test 3 – Filtered Dataset



(g) FFNN test 4 – Filtered Dataset



(h) XGBoost test 4 – Filtered Dataset

Fig. G.4.: Predictions of complex models FFNN and XGBoost on the filtered dataset.

H Real Test Results

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	634.00	197.79
LINEAR-REGRESSION-SIMPLE-FILTERED	457.00	114.65
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	150.00	29.54
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	0.00	100.00
RANDOM-FOREST-SIMPLE-ORIGINAL	239.00	12.26
RANDOM-FOREST-SIMPLE-FILTERED	247.00	16.02
FFNN-SIMPLE-ORIGINAL	3675567.00	1726328.84
FFNN-SIMPLE-FILTERED	2535780.00	1190966.23
FFNN-v1-ORIGINAL	278.00	30.58
FFNN-v1-FILTERED	247.00	16.02
FFNN-v2-ORIGINAL	244.00	14.61
FFNN-v2-FILTERED	253.00	18.84
FFNN-v3-ORIGINAL	194.00	8.88
FFNN-v3-FILTERED	307.00	44.20
FFNN-v4-ORIGINAL	249.00	16.96
FFNN-v4-FILTERED	265.00	24.47
XGBOOST-SIMPLE-ORIGINAL	87.00	59.14
XGBOOST-SIMPLE-FILTERED	105.00	50.68
XGBOOST-v1-ORIGINAL	227.00	6.62
XGBOOST-v1-FILTERED	186.00	12.64
XGBOOST-v2-ORIGINAL	237.00	11.32
XGBOOST-v2-FILTERED	189.00	11.23
XGBOOST-v3-ORIGINAL	135.00	36.59
XGBOOST-v3-FILTERED	167.00	21.56
XGBOOST-v4-ORIGINAL	72.00	66.18
XGBOOST-v4-FILTERED	155.00	27.20

Tab. H.1.: Results of the experiment with input: MSIZE $= 9$, BSIZE $= 500$, NUM_NODES $= 2$. Output: EXEC_TIME $= 212.9$

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	876.00	182.22
LINEAR-REGRESSION-SIMPLE-FILTERED	793.00	155.48
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	811.00	161.28
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	877.00	182.54
RANDOM-FOREST-SIMPLE-ORIGINAL	327.00	5.35
RANDOM-FOREST-SIMPLE-FILTERED	292.00	5.93
FFNN-SIMPLE-ORIGINAL	3675567.00	1184038.85
FFNN-SIMPLE-FILTERED	2535780.00	816839.43
FFNN-v1-ORIGINAL	486.00	56.57
FFNN-v1-FILTERED	456.00	46.91
FFNN-v2-ORIGINAL	380.00	22.42
FFNN-v2-FILTERED	324.00	4.38
FFNN-v3-ORIGINAL	278.00	10.44
FFNN-v3-FILTERED	351.00	13.08
FFNN-v4-ORIGINAL	471.00	51.74
FFNN-v4-FILTERED	386.00	24.36
XGBOOST-SIMPLE-ORIGINAL	391.00	25.97
XGBOOST-SIMPLE-FILTERED	324.00	4.38
XGBOOST-v1-ORIGINAL	382.00	23.07
XGBOOST-v1-FILTERED	348.00	12.11
XGBOOST-v2-ORIGINAL	314.00	1.16
XGBOOST-v2-FILTERED	319.00	2.77
XGBOOST-v3-ORIGINAL	338.00	8.89
XGBOOST-v3-FILTERED	300.00	3.35
XGBOOST-v4-ORIGINAL	336.00	8.25
XGBOOST-v4-FILTERED	349.00	12.44

Tab. H.2.: Results of the experiment with input: MSIZE $= 22$, BSIZE $= 257$, NUM_NODES $= 2$. Output: EXEC_TIME $= 310.4$

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	2459.00	46.53
LINEAR-REGRESSION-SIMPLE-FILTERED	2475.00	46.19
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	3144.00	31.64
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	3085.00	32.92
RANDOM-FOREST-SIMPLE-ORIGINAL	3147.00	31.58
RANDOM-FOREST-SIMPLE-FILTERED	3427.00	25.49
FFNN-SIMPLE-ORIGINAL	3675567.00	79817.53
FFNN-SIMPLE-FILTERED	2535780.00	55035.24
FFNN-v1-ORIGINAL	4010.00	12.81
FFNN-v1-FILTERED	4003.00	12.96
FFNN-v2-ORIGINAL	4294.00	6.64
FFNN-v2-FILTERED	4485.00	2.48
FFNN-v3-ORIGINAL	4746.00	3.19
FFNN-v3-FILTERED	4685.00	1.87
FFNN-v4-ORIGINAL	4334.00	5.77
FFNN-v4-FILTERED	4306.00	6.38
XGBOOST-SIMPLE-ORIGINAL	4587.00	0.27
XGBOOST-SIMPLE-FILTERED	4645.00	1.00
XGBOOST-v1-ORIGINAL	3203.00	30.36
XGBOOST-v1-FILTERED	3418.00	25.68
XGBOOST-v2-ORIGINAL	4524.00	1.64
XGBOOST-v2-FILTERED	4937.00	7.34
XGBOOST-v3-ORIGINAL	4506.00	2.03
XGBOOST-v3-FILTERED	4793.00	4.21
XGBOOST-v4-ORIGINAL	5006.00	8.85
XGBOOST-v4-FILTERED	5108.00	11.06

Tab. H.3.: Results of the experiment with input: MSIZE $= 22$, BSIZE $= 600$, NUM_NODES $= 2$. Output: EXEC_TIME $= 4599.2$

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	3449.00	115.60
LINEAR-REGRESSION-SIMPLE-FILTERED	3692.00	130.79
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	2369.00	48.09
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	2528.00	58.03
RANDOM-FOREST-SIMPLE-ORIGINAL	3776.00	136.04
RANDOM-FOREST-SIMPLE-FILTERED	2175.00	35.96
FFNN-SIMPLE-ORIGINAL	3675567.00	229666.02
FFNN-SIMPLE-FILTERED	2535780.00	158415.97
FFNN-v1-ORIGINAL	3537.00	121.10
FFNN-v1-FILTERED	4026.00	151.67
FFNN-v2-ORIGINAL	1182.00	26.11
FFNN-v2-FILTERED	1541.00	3.67
FFNN-v3-ORIGINAL	1486.00	7.11
FFNN-v3-FILTERED	1622.00	1.39
FFNN-v4-ORIGINAL	1668.00	4.27
FFNN-v4-FILTERED	1547.00	3.29
XGBOOST-SIMPLE-ORIGINAL	2679.00	67.47
XGBOOST-SIMPLE-FILTERED	1598.00	0.11
XGBOOST-v1-ORIGINAL	4004.00	150.30
XGBOOST-v1-FILTERED	3549.00	121.85
XGBOOST-v2-ORIGINAL	3668.00	129.29
XGBOOST-v2-FILTERED	1464.00	8.48
XGBOOST-v3-ORIGINAL	1582.00	1.11
XGBOOST-v3-FILTERED	351.00	78.06
XGBOOST-v4-ORIGINAL	2431.00	51.97
XGBOOST-v4-FILTERED	837.00	47.68

Tab. H.4.: Results of the experiment with input: MSIZE $= 43$, BSIZE $= 345$, NUM_NODES $= 5$. Output: EXEC_TIME $= 1599.7$

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	2139.00	1041.41
LINEAR-REGRESSION-SIMPLE-FILTERED	2275.00	1113.98
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	0.00	100.00
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	0.00	100.00
RANDOM-FOREST-SIMPLE-ORIGINAL	379.00	102.24
RANDOM-FOREST-SIMPLE-FILTERED	291.00	55.28
FFNN-SIMPLE-ORIGINAL	3675567.00	1961248.45
FFNN-SIMPLE-FILTERED	2535780.00	1353037.67
FFNN-v1-ORIGINAL	737.00	293.28
FFNN-v1-FILTERED	803.00	328.50
FFNN-v2-ORIGINAL	219.00	16.86
FFNN-v2-FILTERED	253.00	35.01
FFNN-v3-ORIGINAL	206.00	9.93
FFNN-v3-FILTERED	178.00	5.02
FFNN-v4-ORIGINAL	94.00	49.84
FFNN-v4-FILTERED	231.00	23.27
XGBOOST-SIMPLE-ORIGINAL	275.00	46.74
XGBOOST-SIMPLE-FILTERED	264.00	40.88
XGBOOST-v1-ORIGINAL	388.00	107.04
XGBOOST-v1-FILTERED	481.00	156.67
XGBOOST-v2-ORIGINAL	213.00	13.66
XGBOOST-v2-FILTERED	207.00	10.46
XGBOOST-v3-ORIGINAL	137.00	26.89
XGBOOST-v3-FILTERED	194.00	3.52
XGBOOST-v4-ORIGINAL	89.00	52.51
XGBOOST-v4-FILTERED	207.00	10.46

Tab. H.5.: Results of the experiment with input: MSIZE = 37, BSIZE = 200, NUM_NODES = 6. Output: EXEC_TIME = 187.4

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	5577.00	619.61
LINEAR-REGRESSION-SIMPLE-FILTERED	6153.00	693.94
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	0.00	100.00
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	0.00	100.00
RANDOM-FOREST-SIMPLE-ORIGINAL	1311.00	69.16
RANDOM-FOREST-SIMPLE-FILTERED	670.00	13.55
FFNN-SIMPLE-ORIGINAL	3675567.00	474166.71
FFNN-SIMPLE-FILTERED	2535780.00	327097.42
FFNN-v1-ORIGINAL	2679.00	245.68
FFNN-v1-FILTERED	2333.00	201.03
FFNN-v2-ORIGINAL	758.00	2.19
FFNN-v2-FILTERED	630.00	18.71
FFNN-v3-ORIGINAL	0.00	100.00
FFNN-v3-FILTERED	1426.00	84.00
FFNN-v4-ORIGINAL	1179.00	52.13
FFNN-v4-FILTERED	2126.00	174.32
XGBOOST-SIMPLE-ORIGINAL	665.00	14.19
XGBOOST-SIMPLE-FILTERED	605.00	21.94
XGBOOST-v1-ORIGINAL	2403.00	210.06
XGBOOST-v1-FILTERED	2473.00	219.10
XGBOOST-v2-ORIGINAL	637.00	17.81
XGBOOST-v2-FILTERED	923.00	19.10
XGBOOST-v3-ORIGINAL	715.00	7.74
XGBOOST-v3-FILTERED	832.00	7.35
XGBOOST-v4-ORIGINAL	536.00	30.84
XGBOOST-v4-FILTERED	858.00	10.71

Tab. H.6.: Results of the experiment with input: MSIZE $= 70$, BSIZE $= 200$, NUM_NODES $= 8$. Output: EXEC_TIME $= 775.0$

Model Name	Result	Error%
LINEAR-REGRESSION-SIMPLE-ORIGINAL	1846.00	1508.01
LINEAR-REGRESSION-SIMPLE-FILTERED	2040.00	1677.00
POLYNOMIAL-REGRESSION-SIMPLE-ORIGINAL	1100.00	858.19
POLYNOMIAL-REGRESSION-SIMPLE-FILTERED	641.00	458.36
RANDOM-FOREST-SIMPLE-ORIGINAL	285.00	148.26
RANDOM-FOREST-SIMPLE-FILTERED	213.00	85.54
FFNN-SIMPLE-ORIGINAL	3675567.00	3201613.41
FFNN-SIMPLE-FILTERED	2535780.00	2208767.60
FFNN-v1-ORIGINAL	426.00	271.08
FFNN-v1-FILTERED	1116.00	872.13
FFNN-v2-ORIGINAL	209.00	82.06
FFNN-v2-FILTERED	238.00	107.32
FFNN-v3-ORIGINAL	269.00	134.32
FFNN-v3-FILTERED	71.00	38.15
FFNN-v4-ORIGINAL	175.00	52.44
FFNN-v4-FILTERED	310.00	170.03
XGBOOST-SIMPLE-ORIGINAL	81.00	29.44
XGBOOST-SIMPLE-FILTERED	148.00	28.92
XGBOOST-v1-ORIGINAL	339.00	195.30
XGBOOST-v1-FILTERED	350.00	204.88
XGBOOST-v2-ORIGINAL	225.00	95.99
XGBOOST-v2-FILTERED	230.00	100.35
XGBOOST-v3-ORIGINAL	589.00	413.07
XGBOOST-v3-FILTERED	576.00	401.74
XGBOOST-v4-ORIGINAL	501.00	336.41
XGBOOST-v4-FILTERED	531.00	362.54

Tab. H.7.: Results of the experiment with input: MSIZE $= 35$, BSIZE $= 200$, NUM_NODES $= 13$. Output: EXEC_TIME $= 114.8$