Performance Modeling of Distributed Computing

Master's Thesis of

Valerii Zhyla

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr.-Ing. Anne Koziolek
Second examiner:	Prof. Dr. Ralf H. Reussner
First advisor:	M.Sc. Larissa Schmid
Second advisor:	Dr.-Ing. Tobias Hey

10. November 2023 – 10. Mai 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Optimizing resource allocation in distributed computing systems is crucial for enhancing system efficiency and reliability. Predicting job execution metadata, based on resource demands and platform characteristics, plays a key role in this optimization process. Distributed computing simulators are utilized for this purpose to model and predict system behaviors. Among the various simulators developed in recent decades, this thesis specifically focuses on the state-of-the-art simulator DCSim.

DCSim simulates the nodes and links of the configured platform, generates the workloads according to configured parameter distributions, and performs the simulations. The simulated job execution metadata is accurate, yet the simulations demand computational resources and time that increase superlinearly with the number of nodes simulated.

In this thesis, we explore the application of Recurrent Neural Networks and Transformer models for predicting job execution metadata within distributed computing environments. We focus on data preparation, model training, and evaluation for handling numerical sequences of varying lengths. This approach enhances the scalability of predictive systems by leveraging deep neural networks to interpret and forecast job execution metadata based on simulated data or historical data.

We assess the models across four scenarios of increasing complexity, evaluating their ability to generalize for unseen jobs and platforms. We examine the training duration and the amount of data necessary to achieve accurate predictions and discuss the applicability of such models to overcome the scalability challenges of DCSim. The key findings of this work demonstrate that the models are capable of generalizing across sequences of lengths encountered during training but fall short in generalizing across different platforms.

Zusammenfassung

Die Optimierung der Ressourcenzuweisung in verteilten Computersystemen ist entscheidend für die Verbesserung von Systemeffizienz und Zuverlässigkeit. Die Vorhersage der Metadaten von Jobausführungen, basierend auf Ressourcenanforderungen und Plattformeigenschaften, spielt eine wichtige Rolle in diesem Optimierungsprozess. Die Simulatoren werden eingesetzt, um Systemverhalten zu modellieren und vorherzusagen. Unter den verschiedenen in den letzten Jahrzehnten entwickelten Simulatoren konzentriert sich diese Arbeit speziell auf den Simulator DCSim.

DCSim simuliert die Knoten und Verbindungen der konfigurierten Plattform, generiert die Jobsequenzen gemäß konfigurierter Parameterverteilungen und führt die Simulationen durch. Die simulierten Metadaten von Jobausführungen sind akkurat, doch die Simulationen erfordern Rechenressourcen und Zeit, die überlinear mit der Anzahl der simulierten Knoten steigen.

In dieser Arbeit untersuchen wir die Anwendung von Rekurrenten Neuronalen Netzwerken und Transformer-Modellen zur Vorhersage von Jobausführungsmetadaten in verteilten Computing-Umgebungen. Wir konzentrieren uns auf die Datenvorbereitung, das Modelltraining und die Evaluation zur Handhabung numerischer Sequenzen unterschiedlicher Längen. Dieser Ansatz verbessert die Skalierbarkeit von Vorhersagesystemen, indem tiefe neuronale Netzwerke genutzt werden, um Jobausführungsmetadaten auf der Grundlage von simulierten Daten oder historischen Daten zu interpretieren und vorherzusagen.

Wir bewerten die Modelle in vier Szenarien mit zunehmender Komplexität und untersuchen ihre Fähigkeit, für unbekannte Jobs und Plattformen zu generalisieren. Wir betrachten die Trainingsdauer und die Menge an Daten, die erforderlich ist, um genaue Vorhersagen zu erzielen, und diskutieren die Anwendbarkeit solcher Modelle, um die Skalierbarkeitsprobleme von DCSim zu überwinden. Die wichtigsten Erkenntnisse dieser Arbeit zeigen, dass die Modelle in der Lage sind, über Sequenzen von Längen zu generalisieren, die während des Trainings auftreten. Die Modelle haben aber Schwierigkeiten, über verschiedene Plattformen hinweg zu generalisieren.

Contents

1. Introduction

The Large Hadron Collider (LHC) is the world's largest and most powerful particle accelerator, located at CERN. It consists of a 27-kilometer ring of superconducting magnets and detectors, situated underground near Geneva, Switzerland. The LHC is designed to collide protons or heavy ions at near-light speeds, allowing scientists to explore fundamental questions about the universe, the nature of matter, and the fundamental forces that govern the interactions of elementary particles.

The Large Hadron Collider (LHC) features a particle accelerator ring that is utilized by four primary experimental projects: ATLAS, CMS, ALICE, and LHCb, each operating with distinct detectors and managed by separate organizations Each of these detectors has a specific scientific purpose and is designed to study particular aspects of particle collisions. When the particles collide, they produce many secondary particles. The detectors at the LHC record hits and electrical signals within specific subdetectors, capturing limited-resolution data on certain types of particles based on their interactions with distinct detector materials. From these observations, characteristics such as the energy of some particles, their trajectories through the detector, and the origin of the collision can be reconstructed, albeit with limited precision.

The scale of these experiments results in a massive amount of data. The detectors at the LHC use trigger systems to selectively save data from collisions that are expected to be scientifically significant, filtering out less relevant events to manage the vast amount of data generated. LHC detectors employ a multi-tiered trigger system to decide which data to save. Despite this selective process, the volume of remaining data presents a significant challenge. In November 2022 alone, 26 petabytes of raw experimental data were stored [18].

With the vast amount of data generated by the LHC experiments, there arose a need for a robust computing infrastructure to store, process, and analyze this data. The Worldwide LHC Computing Grid (WLCG) was established to meet this demand. The WLCG integrates thousands of computer centers worldwide into a single, cohesive computing entity to ensure that data from the LHC is readily accessible and analyzable by scientists globally.

Within the WLCG, each particle detector utilizes its own distinct job distribution and scheduling infrastructure, tailored to its specific data processing and analysis requirements. For example, the ATLAS detector employs the PanDA (Production and Distributed Analysis System) for its job scheduling and distributed computing needs [46]. These systems manage the workloads, which consist of jobs related to the corresponding detector, identify suitable data centers, and book resources. The job distribution systems take the data locality into account. When a job requires particular datasets, the job distribution system will try to allocate the job to a grid site where the required data is either already available or can be swiftly transferred. This approach reduces data transfer duration and ensures efficient utilization of network resources.

However, this type of scheduling relies on assumptions about data locations. As the infrastructure and data distribution patterns evolve, these assumptions may no longer hold true in the future. The next generation of Tier-2 sites in several regions will be responsible for computing tasks, but they will not retain any data storage capabilities. To ensure the efficiency of these computations, it will be essential to integrate caching mechanisms at sites with low network bandwidth to the grid storages. The challenging aspect lies in the evaluation of the entire topology, which would require an experimental setup. Implementing such a setup in the real world would be impractical. Hence, there's a need for precise simulators that can simulate job scheduling and execution on a given topology specification.

Many of the current simulators utilized within the WLCG are built upon the GridSim [10, 69] and SimGrid [15] frameworks. GridSim was a popular choice for simulations in distributed computing environments, particularly during the early development of grid computing. Nowadays, SimGrid has taken the forefront, becoming the more widely used tool for contemporary research and simulations in distributed systems.

SimGrid handles higher job loads better than GridSim [23], but both frameworks encounter scalability challenges as the number of simulated entities increases, leading to limitations in their capacity to efficiently process extensive workloads. GridSim exhibits accuracy issues in network simulation, limiting its applicability to simulations of network transfers to some specific scenarios.

SimGrid is a low-level framework designed for detailed simulation of distributed computing environments, including networks, computing resources, and energy consumption. In SimGrid, simulations typically involve the replication of network and system behavior, including data transfer, computation processes, and resource allocation, to study and predict the performance and efficiency of distributed systems [31, 30].

1.1. DCSim Scalability

Distributed Computing Simulator (DCSim) performs accurate simulations of job batches on defined hardware topology, but the simulation time increases super-linear with the number of concurrently executed jobs in the simulation. After the maximal number of concurrently executed jobs is reached, simulation time behaves linearly.

Figure 1.1 presents the required simulation time for different batch sizes of a specific platform. Execution time behaves exponentially from a certain number of jobs, and memory consumption is linear. This behavior of DCSim can be explained by the escalating complexity of the underlying SimGrid simulations. The time required for network simulations in SimGrid grows exponentially with the increase in the number of communicating nodes [23].

The scaling of the execution time below the threshold of full occupation of the CPU cores in simulation is expected to be influenced by the increasing amount of concurrent activities, which need to be handled during simulation [30]. Above this threshold, the behavior is linear driven by the exponential factor.

The amount of concurrent activities is proportional to the number of concurrently active jobs in simulation, creating loads on the simulated platform. With increasing amounts of concurrent activities solving the max-min objective in SimGrid becomes more computationally expensive. Increasing the number of simulated jobs above the threshold of full occupation, the number of concurrent activities creating load on the platform does not increase, since the total number of running jobs at any point in time in simulation is capped and only these jobs create activities during their execution. In simulation, the activities on the platform created by the queued jobs can be neglected compared to the activities introduced by the executing jobs.

Simplifying the platform raises the threshold, yet the exponential behavior persists, which can be seen in Figure 1.1. In this example, the simplification of the platform results in the shift of the threshold from 1.500 jobs to 2.000 jobs. For the real use cases, the simulation of batches with 10.000 to 100.000 jobs is required.

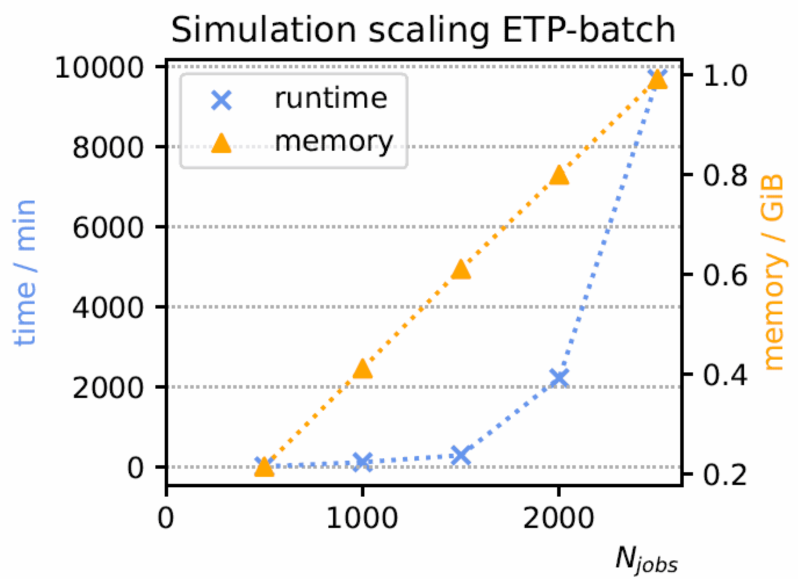


Figure 1.1.: Simulation time on a simplified platform. Figure taken from Horzela [30]

The performance issue with DCSim lies in platform scalability: as the number of simultaneously processed jobs (the active job slots) increases linearly, the simulation time grows exponentially. The memory consumption remains linear. This behavior is presented in Figure 1.2.

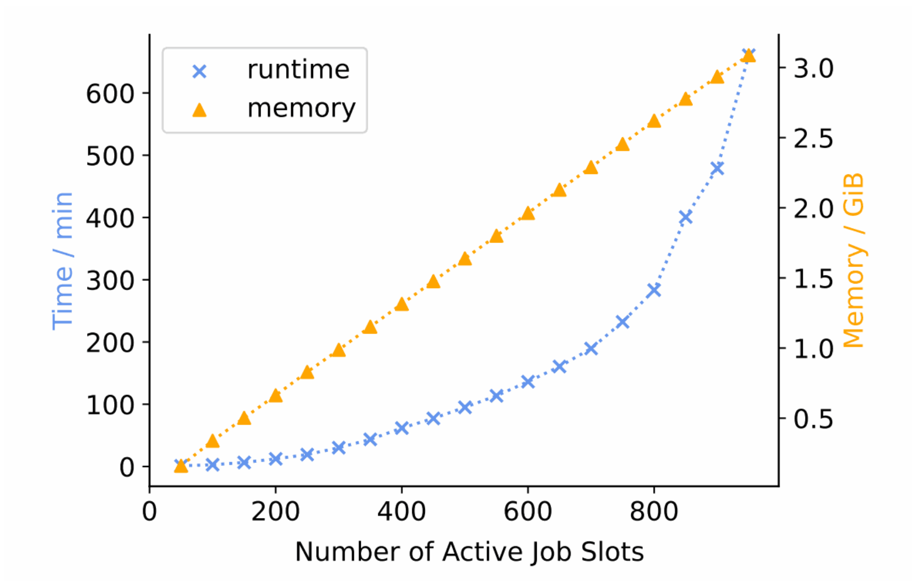


Figure 1.2.: Exponential simulation time growth with a linear increase of job slots number. Figure taken from Horzela [30]

1.2. Objective of this Thesis

Instead of performing time-consuming simulations, we aim to use machine learning models to predict simulation outcomes for each job in the batch. At first, we develop models capable of being trained on shorter job batches and accurately predicting outcomes for job batches with previously unseen lengths. By utilizing a predictive model, the computational costs associated with solving mathematical equations in simulations are eliminated, as the model directly estimates outcomes from learned data. Therefore, our models target two separate tasks: empirically model the DCSim based on training data (data interpolation) and predict the simulation outcomes for longer job batches (data extrapolation). The concept for data preparation and model training is described in Chapter 4, and the experimental design is described in Chapter 5. We evaluate three machine learning architectures in this work - BiGRU, BiLSTM, and Encoder-Only Transformer. Then we incorporate elements of the platform configuration into the model inputs, to address the platform scalability challenge presented in Section 1.1. Our models are trained using data from a diverse array of platforms and subsequently evaluated on platforms they have not previously encountered. The results of incorporation of platform information into the model are described in Chapter 10.

2. Background

This chapter provides a foundational overview of the key concepts, previous research, and theoretical frameworks that are relevant to the study. It contains detailed information about the DCSim simulator, including practical usage examples, an overview of the WLCG infrastructure, a review of various machine learning algorithms and approaches, and a look at performance modeling methodologies.

2.1. WLCG Infrastructure

The Worldwide LHC Computing Grid (WLCG) is a distributed computing infrastructure, developed to store and analyze the extensive data generated by the Large Hadron Collider (LHC) experiments. The data analysis involves the reconstruction of events from the dataset collected directly from the detectors surrounding the place of particle collision, performing the statistical analysis, comparing the data with simulated events, and interpreting data in the context of existing theoretical frameworks. From the abstract perspective, an analysis task can be represented as a batch of computing jobs [20]. Each job has its own specific computational needs, requires a specific data as input, and produces a new data. Most of the jobs are single-core, but some jobs are optimized to run on multiple cores [25].

The WLCG evolved from the conceptual groundwork laid by the Models of Networked Analysis at Regional Centers (MONARC) [43, 55], which provided a foundation for the organization and optimization of data flow and computational tasks for LHC data processing. The structure of WLCG has changed over time, to adapt to the increasing volumes of data produced during the LHC's Run 2 and Run 3, but kept its original design to most parts [6]. The WLCG is structured into a tiered system with four tiers [64].

The Tier-0 center is located at CERN. This is the primary data center where raw data from the LHC experiments is initially processed and then distributed to Tier-1 centers. It also stores a permanent copy of all raw data. Tier-1 centers are major data centers located around the world. They not only store a replicated portion of the raw data from Tier-0 but also handle reprocessing tasks and large-scale analyses. They distribute data to Tier-2 centers and manage data from them. While their primary role is data storage, distribution, and reprocessing, they also facilitate analysis tasks, especially those that require significant computational resources or access to large datasets. Tier-2 centers are responsible for detailed data analysis tasks undertaken by scientists and generation of simulated data. They access the processed data from Tier-1 centers and run analysis jobs that do not require to access raw data. Tier-3 centers are typically smaller local centers or even resources provided by individual university departments. They allow researchers to carry out more specific analyses and are often tailored to the needs of local research groups.

The Grid Computing Centre Karlsruhe (GridKa), located at the Karlsruhe Institute of Technology (KIT), is one of the major Tier-1 centers within the WLCG [37, 53]. It plays an important role in data storage, processing, and distribution for the LHC experiments. As of 2022, GridKa provided 15% of the computing power, 15% of the disk storage, and 13% of the tape storage capacity of the whole WLCG [40].

2.2. DCSim

Distributed Computing Simulator [31] is a tool that generates a batch of jobs and simulates them on arbitrary computing platforms. DCSim is based on the tools Wrench [81] and SimGrid [15].

2.2.1. Configurations

The workload configuration, the dataset configuration, and the platform configuration are the inputs of DCSim simulation. DCSim uses the workload and dataset configurations to generate batches of detailed job characterizations. Listing 2.1 shows an example workload configuration. A workload configuration contains the distributions of job resource consumption values, such as the number of used CPU cores, required floating point operations, volume of RAM, and storage for job output. A parameter can contain a specific numerical value (e.g. "num_jobs" and "submission_time" in Listing 2.1), a histogram description (e.g "cores", "flops", "memory"), a statistical distribution description (e.g "outfilesize"), or a string (e.g "workload_type", "infile_datasets"). The strings are used to represent enum values or filenames. The value of "infile_datasets" is a link to the used dataset configuration. The value "submission_time" is the time at which the entire batch of jobs is submitted. The job batch is considered the smallest unit, all jobs in the batch are submitted simultaneously. The "workload_type" categorizes the jobs into streaming, calculation, and copy jobs. Calculation jobs do not require reading of inputs and focus solely on processing. Copy jobs involve copying input files before processing. Streaming jobs continuously read files as streams while performing calculations.

The structure of the dataset configuration is similar to the workload configuration. Listing 2.2 shows an example dataset configuration. The dataset configuration contains the physical location of a dataset, the number of files contained in the dataset, and the distribution of its respective file sizes. These files are used as job inputs. The value of location should be a valid host, defined in platform configuration.

Listing 2.3 shows an example platform configuration. The listed configuration describes the topology of a KIT network with two computation sites (ETP and GridKa) and the communication gateway between them (KITcentral). Zones defined in configuration

Listing 2.1: Workload Configuration in DCSim

```json
"Analysis_T1": {
    "num_jobs": 32,
    "cores": {
        "type": "histogram",
        "counts": [0, 206746, 35, 0, 3709, 0, 0, 0, 406],
        "bins": [-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5]
    },
    "flops": {
        "type": "histogram",
        "counts": [210623, 263, 4, 6, 0, 0, 0, 0, 0, 0],
        "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15, 1.2e
            +15, 1.35e+15, 1.5e+15]
    },
    "memory": {
        "type": "histogram",
        "counts": [650, 204277, 3161, 1560, 20, 0, 949, 0, 225, 0],
        "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810, 12000]
    },
    "outfilesize": {
        "type": "gaussian",
        "average": 1000,
        "sigma": 0
    },

    "workload_type": "streaming",
    "submission_time": 10,
    "infile_datasets": "Analysis_Dataset_Cache"
}
```

Listing 2.2: Dataset Configuration in DCSim

```json
"Analysis_Dataset_Cache": {
    "location": "GridKA_dCache",
    "num_files": 480,
    "filesize": {
        "type": "histogram",
        "counts": [208159, 967, 701, 447, 270, 154, 72, 31, 38, 6],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
}
```

represent the network zones, that contain different hosts and routes between them. Zones can be connected with other zones, which can be represented with zoneRoutes. Each host contains an id, computation speed in mega floating point operations per second (MFLOPs), number of CPU cores, RAM volume in GiB, and type. Some hosts can also contain different disk volumes, which are characterized by the size in TB, write and read bandwidths in Gigabits per second. In this example, zone ETP contains three hosts - ETPworker, ETPsg0X, and WMSHost. ETPworker is the main worker node of ETP, it processes the scheduled jobs. ETPsg0X is the main storage node of ETP, which is called a cache in this context. WMSHost is the scheduler node of ETP, it processes the job requests and assigns jobs to the worker node. These hosts are interconnected with links, that have specific bandwidth values in and Gigabits per second latency values in microseconds. A link represents a physical link, which can be used for data transfer between nodes. GridKA is the second zone in this network. For simplicity, only the gateway of this zone is presented in Listing 2.3. Both zones ETP and GridKA are interconnected via the third zone KITcentral. The gateway router of KITcentral is connected to the gateway router of ETP and to the gateway router of GridKa. We can use such platform configurations to describe platforms of arbitrary scale, from small single-node workers to the whole WLCG network.

Listing 2.3: Part of KIT Platform Configuration in DCSim

```xml
<platform>
    <zone id="KIT" routing="Floyd">
        <!-- ETP -->
        <zone id="ETP" routing="Floyd">
            <!-- ETP hosts -->
            <host id="ETPworker" speed="4476Mf" core="448">
                <prop id="type" value="worker"/>
                <prop id="ram" value="960GiB"/>
            </host>

            <host id="ETPsg0X" speed="2726Mf" core="60">
                <prop id="type" value="worker,cache"/>
                <prop id="ram" value="160GiB"/>
                <disk id="ssd_cache1" read_bw="9.6Gbps" write_bw="9.6Gbps">
                    <prop id="size" value="7.5TB"/>
                    <prop id="mount" value="/"/>
                </disk>
            </host>

            <host id="WMSHost" speed="10Gf" core="10">
                <prop id="type" value="scheduler,executor"/>
                <prop id="ram" value="16GB"/>
            </host>

            <!-- ETP routing -->
            <router id="ETP_gateway"/>

            <link id="etp_sched" bandwidth="10Gbps" latency="0us"/>
            <link id="etp_worker_FATPIPE" bandwidth="10Gbps" latency="0us"/>
            <link id="etp_worker" bandwidth="80Gbps" latency="0us"/>
            <link id="etp_sg0X_FATPIPE" bandwidth="10Gbps" latency="0us"/>
            <link id="etp_sg0X" bandwidth="40Gbps" latency="0us"/>

            <route src="ETP_gateway" dst="WMSHost">
                <link_ctn id="etp_sched"/>
            </route>

            <route src="ETP_gateway" dst="ETPworker">
                <link_ctn id="etp_worker_FATPIPE"/>
                <link_ctn id="etp_worker"/>
            </route>

            <route src="ETP_gateway" dst="ETPsg0X">
                <link_ctn id="etp_sg0X_FATPIPE"/>
                <link_ctn id="etp_sg0X"/>
            </route>
        </zone>

        <!-- GridKA site description (shortened) -->
        <zone id="GridKA" routing="Floyd">
            <!-- GridKa hosts... -->
            <!-- GridKa routing...-->
            <route src="GridKa_gateway" dst="TOPAS">
                <link_ctn id="gridka_topas"/>
            </route>
        </zone>

        <!-- KIT Gateway that connects ETP and GridKA -->
        <zone id="KITcentral" routing="Floyd">
            <router id="KIT_gateway"/>
            <link id="etp_to_gridka" bandwidth="100Gbps" latency="0us"/>
        </zone>

        <zoneRoute src="ETP" dst="KITcentral" gw_src="ETP_gateway" gw_dst="
            KIT_gateway">
            <link_ctn id="etp_to_gridka"/>
        </zoneRoute>

        <zoneRoute src="GridKA" dst="KITcentral" gw_src="GridKa_gateway" gw_dst="
            KIT_gateway">
            <link_ctn id="etp_to_gridka"/>
        </zoneRoute>
    </zone>
</platform>
```

2.2.2. Simulation Input and Output

Listing 2.4 contains an example command that starts a DCSim simulation with a specific platform configuration, workload configuration, and dataset configuration, and saves the simulation result in file example-output.csv.

Listing 2.4: DCSim simulation start command

```
dc-sim --platform data/platform-files/WLCG_disklessTier2.xml
    --workload-configurations data/workload-configs/T2_DE_DESY_workloads.json
    --dataset-configurations data/dataset-configs/prefetchScan.json
    --output-file example-output.csv
```

Each simulation requires a seed for a random number generator. This seed can also be set externally with –seed option. Listing 2.5 shows an example of the usage of DCSim with a specific seed.

Listing 2.5: DCSim simulation start command with a specific seed

```
dc-sim --platform data/platform-files/WLCG_disklessTier2.xml
    --workload-configurations data/workload-configs/T2_DE_DESY_workloads.json
    --dataset-configurations data/dataset-configs/prefetchScan.json
    --output-file example-output.csv
    --seed 424242
```

Listing 2.6 contains a part of file "example-output.csv". Each simulated job is represented as an entry in the CSV format. The entry contains both simulation input parameters (job.tag, job.flops, infiles.size, outfiles.size) and simulated outputs (machine.name, hitrate, job.start, job.end, job.computetime, infiles.transfertime, outfiles.transfertime)

Listing 2.6: DCSim simulation result

```
job.tag, machine.name, hitrate, job.start, job.end, job.computetime, job.flops,
    infiles.transfertime, infiles.size, outfiles.transfertime, outfiles.size
job_Analysis_T2_31, Tier128, 0, 10.000325, 62.912420, 24.657514,
    157512196845.572174, 36.559648, 41993201318.724899, 0.199152, 612605106.913359
job_Analysis_T2_37, Tier128, 0, 10.000303, 75.188354, 36.900974,
    235723420983.291718, 37.486082, 44994464536.159149, 0.255969, 1241127247.730922
```

2.2.3. Job Input and Output Parameters

Table 2.1 describes the input parameters of each job in simulation. Additionally, we can use the position of the job in batch. Table 2.2 describes the output parameters of each job in simulation.

Input Parameter	Description
Floating Point Operations	Volume of floating-point operations the job undertakes during its execution. Measure of the computational workload of the job.
Input Files Size	Volume of data that the job takes as input. Measured in bytes.
Output Files Size	Volume of data that the job produces as a result after processing. Measured in bytes.

Table 2.1.: Simulation input parameters

2.2.4. Representation of Simulation Results

One DCSim execution with fixed seed, platform configuration, workload configuration and dataset configuration generates a CSV file, wherein each line corresponds to a simulated job. The DCSim configuration files and simulation outputs are described in Section 2.2. The Listing 2.6 presents a small example of such CSV file, the Section 2.2.3 contains a description of job input and output parameters.

We refer to such a CSV file as a single simulation result, and we call the sequentially arranged list of jobs within it a job batch. Each simulation has a unique id. Within the simulation, we sort the jobs in ascending order based on their 'job_start' values and assign an index for each job. The job with the smallest value of 'job_start' is assigned an index of 1, and the job with the greatest value of 'job_start' is assigned an index of the length of the batch. If two or more jobs have the same value of 'job_start', the jobs are sorted by job name, which is always unique in DCSim.

We choose this representation because it allows our models to utilize the actual order of job execution as provided, without attempting to emulate the behavior of the job scheduler. The focus should be on analyzing the sequence of job executions as they are, rather than on deciphering the scheduling strategy, especially at the outset when the system's capacity for new jobs is not yet fully occupied.

In the training and evaluation datasets, results from multiple simulations are concatenated. This structural arrangement of an abstract dataset with two simulations is schematically illustrated in Figure 2.1. The first simulation contains 1000 jobs, and the second simulation contains 2500 jobs.

Opting for this approach reduces the number of files to handle, as it consolidates the data from several simulations into a single file, rather than maintaining a one-to-one correspondence between files and simulations. Should the file size become too large for memory allocation, alternative strategies like streaming the data or splitting the large dataset into several smaller files can be employed.

In the experiments conducted for this work, datasets comprising 10.000 or fewer simulations were utilized in each scenario. Such datasets typically result in files that are

Output Parameter	Description
Start Time	The point at which a job is submitted into the system for processing.
End Time	The moment when a job has been fully processed and the results are made available.
Computation Time	The actual time the job was actively being computed by the CPU, excluding any waiting or idle periods.
Machine name	Identifier of host where the job was processed.
Hit Rate	Proportion of input files that are already available in the storage of the computational environment, eliminating the need for data transfer from remote storage.
Input Files Transfer Time	Duration taken to transmit and store the necessary input files from their source location to the designated computational environment.
Output Files Transfer Time	Duration taken to transmit and store the results of a job, once processed, from the computational environment to the desired storage location.

Table 2.2.: Simulation output parameters

approximately 1GB in size, offering a manageable balance between data volume and file size.

2.3. Performance Modelling

Performance modeling is concerned with the description, analysis, and optimization of the dynamic behavior of computer and communication systems [28]. This involves the investigation of the flow of data, and control information, within and between components of a system and subsequent processing. The aim is to understand the behavior of the system and identify the aspects of the system which are sensitive from a performance point of view [28].

Predictive performance modeling specifically refers to the use of performance models to predict future performance based on historical data and trends [76]. It often relies on machine learning or statistical methods to make predictions about future system states or to forecast the impact of changes to the system. Predictive models are trained using a set of input data that reflects the various factors affecting performance, such as workload characteristics, system configurations, and operational parameters. Once trained, these models can predict outcomes such as response times, system throughput, and resource utilization [74].

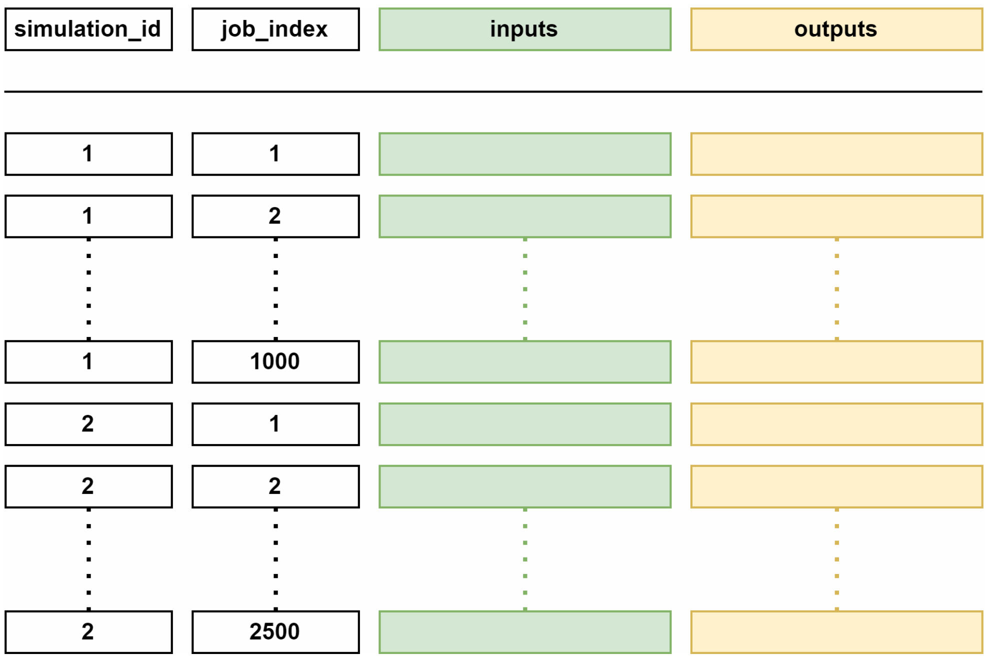


Figure 2.1.: Representation of simulations in the dataset

2.3.1. Analytical Performance Modelling

Analytical Performance Modeling is a method used to create mathematical representations of a system to predict its performance. The analytical performance model is a function $f(x_1, x_2 ... x_n) = y$, where x_i is one of n parameters and y is a predicted outcome [11]. The outcome can be a single value or a vector of values. For example, a performance model can be a function that takes the resource description and characterization of workload and returns some metrics such as response time or resource utilization [74].

2.3.2. Empirical Performance Modelling

Empirical Performance Modeling is an approach to understanding system performance by relying on observation and measurement of the system's behavior under various conditions. Unlike analytical models, which are grounded in theory and use mathematical formulations to predict performance, empirical models are derived from actual data collected from the system in operation. Empirical performance modeling can be useful in scenarios where theoretical models are too complex or when insufficient knowledge about the system's inner workings is available [33]. Machine learning techniques, ranging from regression analysis to advanced deep learning methods, are frequently employed in empirical performance modeling [76, 48].

Empirical performance models can predict the performance of algorithms on previously unseen input, including previously unseen problem instances or previously untested parameter settings. Such models are useful for analyzing how an algorithm performs under different conditions, for selecting samples for a new problem instance, or for creating the surrogate models [2].

2.4. Machine Learning

This section provides a overview of the machine learning concepts and model architectures employed in this study.

2.4.1. Interpolation vs Extrapolation

Interpolation is the process of estimating unknown values within the range of a discrete set of known data points [34]. Interpolation is often used when the data has missing values or gaps, and you want to fill in those gaps with plausible data that fits the sequence's existing trend or pattern. In the use case described in this work, interpolation refers to the prediction of simulation results for job batches of lengths that fall within the range of those used during the model's training and validation phases.

Extrapolation is the process of estimating values outside the range of known data points [41]. Unlike interpolation, where the predictions are made within the bounds of the existing data, extrapolation extends the sequence beyond the known data points, either forward or backward. In the use case described in this work, extrapolation refers to the prediction of simulation results for job batches of lengths that extend beyond the range of those used during the model's training and validation phases.

2.4.2. Model Training

Model training is the process of teaching a machine learning model to make predictions or decisions based on input data by adjusting its parameters to minimize the difference between the predicted values and the actual values. An epoch during training refers to one complete pass through the entire training dataset, where the model has had the opportunity to learn from every example provided. Many epochs are used during training to ensure that the machine learning model has sufficient exposure to the entire training dataset, allowing it to iteratively adjust its parameters and improve its prediction accuracy over time.

A model hyperparameter is a configuration that is external to the model and whose value is set before the training process begins. It influences the structure of the model and how the model is trained, but it does not change during the training process itself. Hyperparameters include settings such as the learning rate and the number of layers. The learning rate, in particular, is a crucial hyperparameter that determines the size of the steps the model takes during optimization to minimize the loss function, directly impacting the convergence and performance of the model. Hyperparameter optimization is crucial in machine learning as it systematically identifies the optimal settings that enhance model accuracy and generalization, directly impacting the model's efficacy on unseen data.

2.4.3. Surrogate Models

A surrogate model is a simplified representation or approximation of a complex real-world system or process. The idea is to use the surrogate model in place of the more complex, detailed model, especially in scenarios where numerous simulations are required. Such models are often derived from a set of data obtained from detailed simulations or experiments. Surrogate models are commonly employed in general-purpose simulation frameworks [27] and specialized tools designed for specific physics domains, such as fluid simulations [8], where they enhance computational efficiency.

Surrogate modeling can be utilized to overcome the computational limitations of simulators by providing faster approximations that significantly reduce the need for computationally intensive simulations. By training on data from detailed simulations or experiments, surrogate models can predict outcomes with reasonable accuracy, enabling more extensive exploration and optimization of the system or process with a fraction of the computational resources.

2.4.4. RNN

An RNN (Recurrent Neural Network) is a class of artificial neural networks designed to recognize patterns in sequences of data, such as text, genomes, or numerical time series [50]. Its purpose is to process sequential information, where the output from previous steps is used as input for the current step, making it capable of dynamic temporal behavior.

An RNN works by maintaining a hidden state that captures information from previous inputs in the sequence. At each step in the sequence, it combines the current input with its current hidden state to produce an output, which is then passed along with the next input into the model for the subsequent step, enabling it to make decisions based on the accumulated knowledge of the sequence so far.

RNNs are widely applied in fields such as natural language processing for tasks like language translation and sentiment analysis and in time series analysis for forecasting stock prices or weather patterns. However, RNNs face challenges with long sequences due to vanishing and exploding gradient problems, which can hinder their ability to learn dependencies over long distances [60].

2.4.5. LSTM

An LSTM (Long Short-Term Memory) network is a type of RNN that is specifically designed to address and overcome the vanishing and exploding gradient problems inherent in traditional RNNs [57]. Its primary purpose is to capture long-term dependencies in sequence data more effectively than standard RNNs, enabling it to remember information for extended periods. LSTMs are extensively used in applications requiring the analysis of sequences with long-range temporal dependencies, such as in natural language processing for tasks like text generation and machine translation, and in time series prediction for financial forecasting or weather prediction.

The key innovation of LSTMs is their use of gated units, including input, output, and forget gates, which regulate the flow of information in and out of the cell, preventing gradients from vanishing or exploding during backpropagation [29]. This architecture allows LSTMs to learn from data where relationships span over long sequences, significantly enhancing their applicability to a wide range of sequence learning tasks compared to traditional RNNs. Although the concept of LSTM was first presented in 1997, various modifications of them are actively utilized in both research and production environments today.

Standard LSTM processes data in a single direction, typically forward, from the beginning of the sequence to the end. It can only utilize the information from previous elements in the sequence when making predictions. BiLSTM (Bidirectional LSTM) model processes data in both forward and backward directions. It has two sets of LSTM layers, one for each direction. By processing the sequence in both directions, a BiLSTM can capture context from both past and future elements in the sequence. This dual-direction processing makes BiLSTMs effective for tasks where understanding the entire context of the input sequence is crucial.

2.4.6. GRU

A Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture designed to solve the vanishing gradient problem present in traditional RNNs by more effectively capturing dependencies for sequences of data. Its purpose is to simplify the LSTM model while retaining its ability to remember information over long periods, making it more computationally efficient [78]. They incorporate a gating mechanism, consisting of reset and update gates, which manages the flow of information without the need for a separate memory cell, unlike LSTMs [21]. This streamlined architecture allows GRUs to perform on par with LSTMs on various tasks with fewer parameters and computational overhead, addressing both the vanishing and exploding gradient issues effectively [57].

The primary difference between BiGRU (Bidirectional GRU) and GRU is that BiGRU processes data both forwards and backwards to capture context from both directions of the sequence, while GRU processes data sequentially in a single direction. This bidirectional processing in BiGRU is achieved by utilizing two GRU layers in parallel, one reading the sequence from start to end and the other from end to start, and then combining their outputs.

2.4.7. Transformer

A Transformer is a deep learning model that relies on a self-attention mechanism to process sequential data, allowing it to weigh the importance of different parts of the input data irrespective of their positions [71]. Its primary purpose is to handle sequences without the need for recurrent architecture, enabling parallel processing and significantly reducing training times [54].

The Transformer model features an encoder-decoder structure. The encoder maps the input sequence to a high-dimensional representation, leveraging self-attention mechanisms to capture global dependencies. The decoder generates the output sequence, using both its self-attention and the encoder's output to focus on relevant parts of the input.

Initially introduced for natural language processing (NLP) tasks, such as translation and text summarizing, Transformers are now being adapted for numerical tasks such as time series forecasting, where their ability to capture long-range dependencies in data can be leveraged to predict future values based on past observations.

3. Related Work

This chapter presents an overview of the key research in the areas of grid computing simulators and empirical performance modeling using machine learning techniques.

3.1. Simulators used in Grid Computing

GridSim [10, 69, 47] and SimGrid [12, 13, 15, 42] are the two most widely used frameworks for WLCG simulations, but SimGrid has become the more prevalent choice in recent times. The scalability of grid simulation frameworks was compared by W. Depoorter et al. [23]. This comparison included the GridSim, SimGrid, and many other simulators, including older ones. This comparison shows, that SimGrid handles higher job loads better than GridSim [23], but both frameworks encounter scalability challenges as the number of simulated entities increases. WRENCH (Workflow Research ENabling Cyberinfrastructure for High-throughput) [14] is an open-source framework, built on top of SimGrid. WRENCH provides directly usable high-level simulation abstractions, to make it possible to implement simulators of complex scenarios with minimal development effort. The simulator DCSim [31], which we aim to improve with surrogate models, is based on WRENCH.

René Caspart et al. [16] made a first attempt to use the Palladio Simulator [5] to evaluate different strategies for utilization of computing resources in WLCG.

3.2. Machine Learning in Performance Modeling

Jingwei Sun et al. [70] discuss the development of an automated method for modeling and predicting the performance of HPC applications using machine learning techniques. The authors address the complexity of accurately predicting the performance of parallel programs due to the influence of various factors such as hardware, applications, algorithms, and input parameters. They propose a method that involves automatically instrumenting an MPI program to output a feature vector and corresponding execution time after each run. Using a random forest machine learning approach, they build an empirical performance model that can predict execution times for new inputs. They also introduce a transfer learning method to improve prediction accuracy on new platforms without extensive historical execution data.

Preeti Malakar et al. [48] present a benchmark study evaluating eleven ML methods for modeling the performance of four scientific applications on four HPC platforms. The results indicate that bagging [9], boosting [9], and deep neural networks are promising ML methods for this task, and that transfer learning with deep neural networks can significantly improve cross-platform performance prediction. We are focusing on deep neural networks for performance prediction, and this work provides insights about transfer learning, which are important for the generalization of our model.

Carl Witt et al. [76] provide a review of performance prediction methods for batch processing in distributed systems. The authors analyzed different predictive performance modeling approaches that predict performance metrics like execution duration, required memory, or wait times for future jobs and tasks based on past performance observations. The paper classifies and compares sources of performance variation, predicted performance metrics, limitations and challenges, required training data, use cases, and prediction techniques.

3.3. Relevant Machine Learning Architectures

Hojjat Salehinejad et al. [61] provide a comprehensive survey on RNNs and discuss several new advances in the field. The authors explain the architecture of RNNs and the gradient-based learning methods, including backpropagation through time (BPTT), and the importance of proper initialization of weights and biases for effective training. The paper covers regularization methods for RNNs to prevent overfitting and compares different gradient descent methods for optimizing RNNs. The major advances in RNNs from 1990 to 2017 are also presented in this paper, such as the introduction of LSTM and GRU networks, and various optimization techniques.

3.3.1. BiLSTM

Khaled A. Althelaya et al. [39] compared the performance of LSTM and BiLSTM architectures for short-term and long-term predictions of stock prices. Sima Siami-Namini et al. [65] conducted an analysis of various LSTM architectures for forecasting the time series of stock prices. Both works indicated that BiLSTMs generally outperform standard LSTMs in time series forecasting.

3.3.2. BiGRU

Kamal Basulaiman and Masoud Barati [38] used the BiGRU-based model for power system state forecasting. They employed the BiGRU layers for capturing long-term dependencies and the 1D convolution layers for feature extraction.

3.3.3. Transformer

Ashish Vaswani et al. [71] introduced the Transformer model, which is based on a self-attention mechanism that directly models relationships among all words in a sentence, regardless of their positions. Transformers were initially designed for Natural Language Processing tasks, but this architecture also found application in sequence-to-sequence prediction. In our work, we will utilize the Transformer model for predictions. Neo Wu et al. [77] present an approach that utilizes Transformer models for time series forecasting.

Jacob Devlin et al. [35] introduced BERT (Bidirectional Encoder Representations from Transformers). BERT works by pre-training a deep bidirectional Transformer encoder on a large corpus of text, using tasks like masked language modeling and next sentence prediction to understand the context of words from both directions. Once pre-trained, BERT can be fine-tuned with additional output layers.

Yann Dubois et al. [24] address the problems of neural networks in extrapolating patterns beyond the data they have been trained on. Despite the proficiency of neural networks in interpolation they often struggle with extrapolation, particularly when the required abstractions are simple. The authors propose that Transformer models equipped with a separate content- and location-based attention mechanism are more likely to successfully extrapolate to longer sequences than models with standard attention mechanisms.

3.4. Similar Application Fields

Numerous studies employ various RNN architectures for modeling the dynamics of numerical time series forecasting. Sreelekshmy Selvin et al. [62] applied a sliding window approach for predicting future stock prices with LSTM. Amit Joshi et al. [63] compared the LSTM, GRU, SVM (Support Vector Machine) and MLP (Multilayer Perceptron) models with different metrics, including R-squared, for stock price prediction. The LSTM and GRU models demonstrated the best results. Siddartha Mootha et al. [52] performed a sequence-to-sequence modelling of stock prices with BiLSTM model. Yuanshuai Duan et al. [80] applied different BiGRU modifications to model stock price dynamics.

4. Concept

The data relationships in distributed computing are complex, and data has too high dimensionality for straightforward statistical models. The sequence length is potentially unlimited, which requires either the segmentation of data into parts with fixed numbers or elements or employing model architectures that can process arbitrary sequences. Both the simulated jobs and the simulated platform are dynamic, and different parts of simulation data can be generated by different processes.

In this work, we investigate the capabilities of deep neural networks in predicting the job execution metadata in distributed computing systems and utilize these networks to address the DCSim scalability challenges outlined in Section 1.1. A high-level overview of the data generation and model training process is presented in Figure 4.1. Initially, we conduct simulations and preprocess the resulting data. This preprocessed data is then utilized for both training and evaluating the model. Once the model has been trained, it is employed to predict outcomes for new job sequences, which are several times longer than the sequences used during the training phase. The same models and concepts can be applied to real-world data instead of simulated data since DCSim uses the same format for job inputs and outputs as WLCG logs.

We developed a methodology for numerical sequence-to-sequence predictions across sequences of varying lengths, which is adaptable for use in various contexts. This chapter describes the concepts used for data preparation, model training, and evaluation of model predictions. This methodology is applicable to any sequence of objects, enabling the prediction of certain attributes based on other attributes within the object, all within the context of a predefined fixed-length segment of the sequence.

4.1. Data Preprocessing

In DCSim output, each object is a simulated job, and a sequence is a batch of sequentially simulated jobs. The representation of DCSim simulations is described in Section 2.2.4. Once the simulations have been conducted and the dataset for some specific platform is created, we preprocessed the data to split it into training and evaluation datasets, normalize, and cut it into segments of the same length. During model training, the training dataset is shuffled after each epoch.

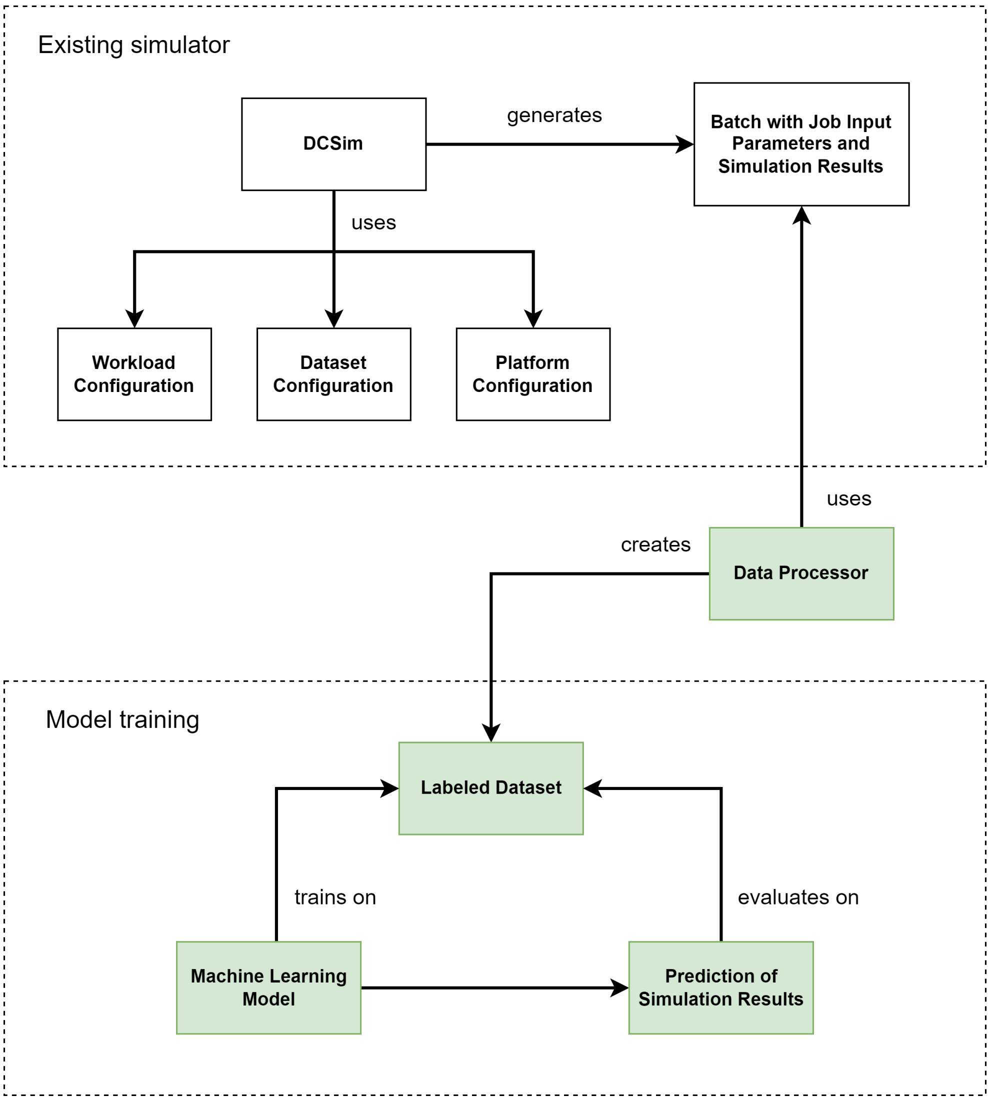


Figure 4.1.: Model training and evaluation using DCSim simulations data

4.1.1. Data Split

We divide the dataset two distinct subsets: the training set and the evaluation set. Each set should contain distinct simulations and represent simulations of various lengths to ensure a comprehensive and diverse range of data for both training and evaluation purposes, with no overlap between the training and evaluation datasets.

In this work, we adhere to a 70/30 split for dividing the dataset into training and evaluation sets [72]. At first, we calculate the number of simulations of each length present in the dataset. Subsequently, we allocate 70% of the simulations from each length category to the training dataset. The remaining 30% of the simulations are then saved as the evaluation dataset.

4.1.2. Data Standardization

We standardize the data by subtracting the mean and dividing by standard deviation to ensure that all features contribute equally to the regression model's predictions, preventing features with larger scales from dominating the learning process [58]. Standardization also improves the numerical stability of optimization algorithms, leading to faster convergence and more accurate regression models.

This process entails two steps: first, fitting the scaler to the data, and then applying the fitted scaler to the data. There are two approaches to consider: employing an individual scaler for each feature (for each input and output parameter) or using a single scaler for all features. A single scaler can be applied to all features when they are already on similar scales in the dataset, standardizing them without distorting their relationships. In our use case, each feature can exhibit different ranges and distributions. Employing a separate scaler for each feature allows for scaling that is tailored to the statistical properties of each feature, adjusting the mean to zero and the standard deviation to one, without being influenced by the scales of other features. Therefore, we have chosen to use this method for our data standardization process.

Incorporating knowledge about the data's distribution is crucial for selecting the appropriate scaler, as the effectiveness of different scaling methods varies with the underlying characteristics of the dataset [1]. Since all parameter values generated by DCSim adhere to a normal distribution, we employ a StandardScaler [67] for data standardization of each feature.

Once applied, the scaler is retained in memory to facilitate the inverse scaling of the predicted data. While standardized data is instrumental in calculating various metrics, the ultimate objective is for the output of our model to mirror the range and distribution of the actual DCSim simulation results. Therefore, it is essential to reverse the standardization process on the model's output to align the predicted values with the real-world scale and distribution of the simulation results. This step ensures that our model's predictions are practically applicable and comparable to the original DCSim results.

4.1.3. Data Windowing

Some simulations in our dataset contain only a single job, while others can encompass up to 100,000 jobs. Real application scenarios can go far beyond this range. To accommodate the wide variation in simulation sizes, it is necessary to align the data to a common size because LSTMs and GRUs require fixed-length input sequences to maintain consistent internal state dimensions and ensure the proper functioning of their recurrent architecture. This alignment enables a single model to predict the results of simulations regardless of their differing sizes by breaking down variable-length simulations into uniformly sized chunks that the model can process.

In our study, we considered two methods to align the data: the first is to pad all sequences with zeros until they match the length of the longest sequence in the dataset, and the second is to split each simulation into smaller, fixed-size segments, known as windows [45]. The padding becomes impractical and inefficient when dealing with sequences of potentially unlimited length, therefore we decided to use the data windowing.

If the sequence does not align perfectly with the window sizes, the final window is padded with zeros. For instance, consider a scenario where we want to create a window sequence for two simulations in the dataset, first with 5 jobs, and second with 4 jobs. The window size is set to 3 and the window overlap is 0. A simulation with 5 jobs can be split into two windows, the second window will be padded with one entry. A simulation with 4 jobs can be split into two windows, the second window will be padded with two entries. This example is depicted in Figure 4.2.

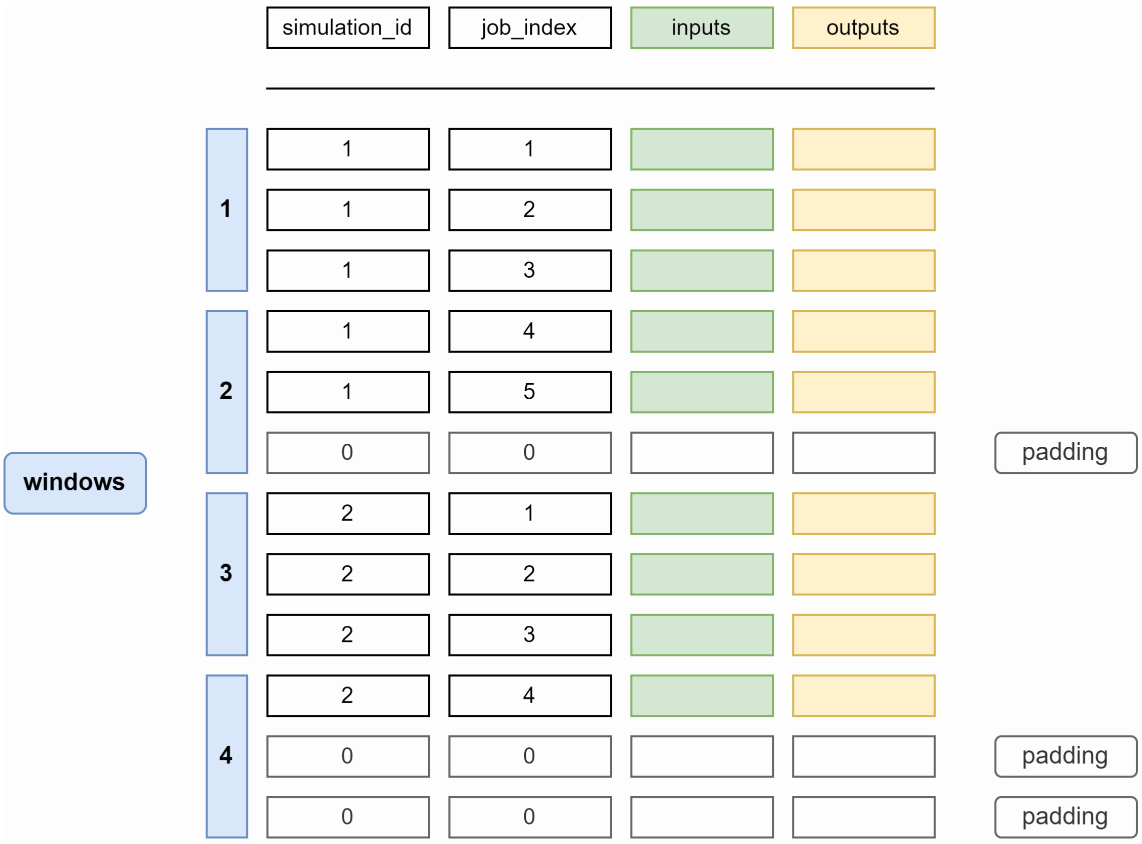


Figure 4.2.: Simulations as a sequence of windows without overlap

Models in this work have the ability to retain information across time steps within a single window due to their internal memory cells. However, this memory is reset between different sequences. Windows overlap for consecutive windows within the same simulation can be used to address this issue. This design ensures that data from distinct simulations are segregated, maintaining the integrity of each simulation's dataset. Overlapping windows could be potentially useful for capturing the continuity and dependencies between jobs within the same simulation, enhancing the model's ability to learn from sequential patterns. It is important to ensure that an overlap only occurs within the data from the same simulation. The data from different simulations should not be mixed in the same window, the model is designed to make a separate prediction for each simulation. Figure 4.3 illustrates this concept using the same sequences detailed in the previous example, with the introduction of a one-job overlap between windows.

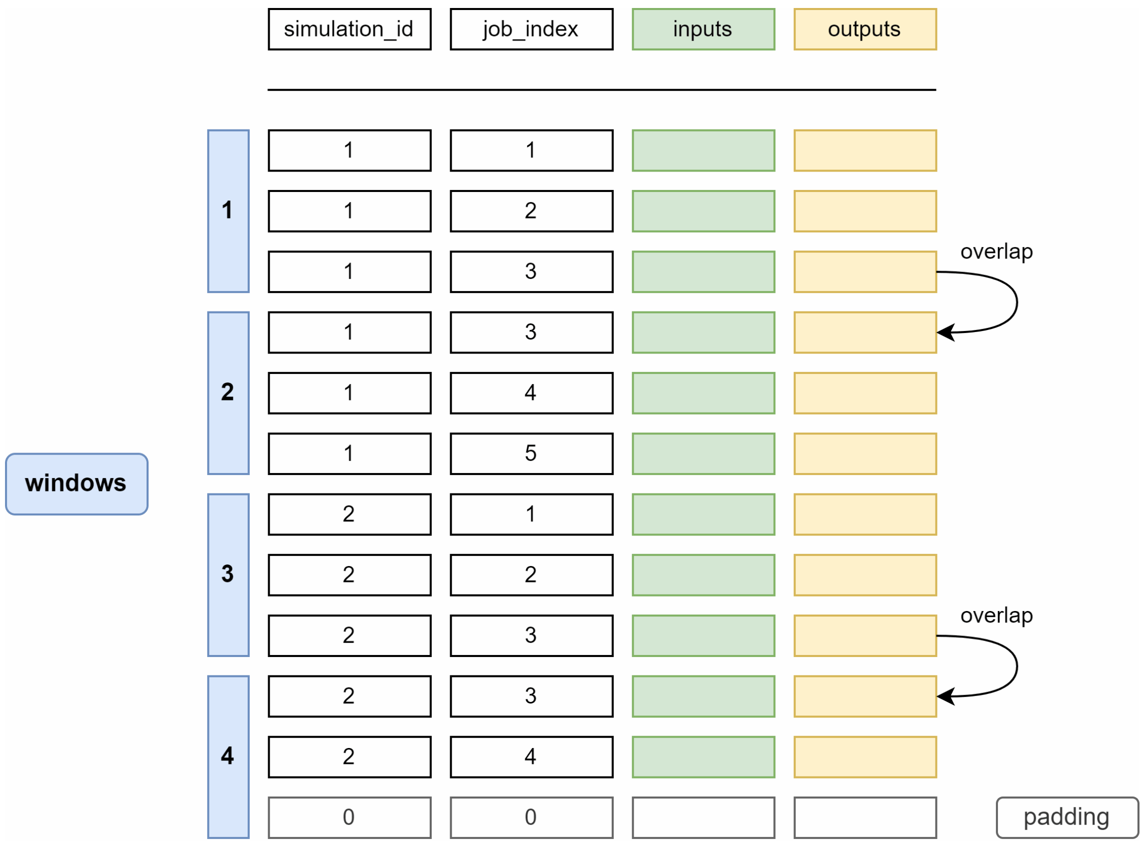


Figure 4.3.: Simulations as a sequence of windows with overlap

The edge case for overlap in this context occurs when the overlap is set to $(n - 1)$ for a window of size n. This scenario essentially mirrors the concept of a rolling window, where each new window shifts by just one job from the preceding window. Although this approach could provide a high-resolution view of the simulation data, enabling detailed sequential analysis, but increases memory demands from $O(n)$ to $O(n^2)$. Implementing this method requires additional optimizations to efficiently manage memory usage and ensure the scalability of the data processing pipeline. In this work, models that used large overlaps did not demonstrate superior predictive capabilities, leading us to decide against investing time in these optimizations.

4.1.4. Shuffling

Shuffling the windows in the training dataset after each epoch helps to prevent overfitting by ensuring the model cannot memorize the order of data, thereby improving its ability to generalize to unseen data [56]. It breaks correlations between consecutive samples, encouraging the model to learn from the data's intrinsic characteristics rather than its sequence. This technique also promotes a more dynamic learning process, aiding in the balanced exposure of the model to diverse data features and potentially reducing biases.

However, employing shuffling can complicate our windowing approach, which converts a sequence of simulations into a plain sequence of windows. If the windows are shuffled and lack an identifier of the original sequence, reconstructing the correct order after model prediction will not be possible. To circumvent this issue, we have included 'simulation_id' as part of the input data, which is a unique identifier for each sequence. We generate the 'simulation_id' during the dataset preparation. Incorporating 'simulation_id' and 'job_index' into the input windows enables us to reconstruct the original data sequence and eliminate duplicates arising from overlapping windows, facilitating the cleansing of data before evaluation. Only the 'job_index' is used for model training as an input parameter, the 'simulation_id' serves as metadata and is not used by the model directly.

4.2. Model Architecture

The objective of our models is to predict the outcomes of each simulated job based on its input parameters and the context provided by other jobs within the same sequence. In our research, we are exploring the capabilities of three different model architectures. All three models are designed to take a window as input, which contains a sequence of jobs from a single simulation. They process this input to compute a new window, maintaining the same number of entries, wherein each entry corresponds to the values of output parameters for each job.

The 'GRU' model comprises an input linear layer, multiple BiGRU layers, and a single output linear layer [61, 80]. The number of BiGRU layers and the hidden size within these layers are free parameters. The input layer maps the original input parameters to the specified hidden size, ensuring compatibility with the BiGRU layers processing. These BiGRU layers then operate in the hidden size domain, effectively processing and transforming the data through each layer. Finally, the outputs from the last BiGRU layer are mapped back to the dimensions of the output parameters, thereby producing the model's predictions. The GRU model is described in Section 2.4.6, and the BiGRU modification is described in Section 2.4.6. The primary difference between GRU and BiGRU is their approach to processing sequential data: GRU processes data in a forward direction, making it suitable for tasks where future context is not essential, whereas BiGRU processes data both forward and backward, making it more effective in scenarios where the context from both directions is important for understanding the sequence. In simulations, future jobs cannot influence the outcomes of already completed jobs. However, during the training phase, the model could potentially benefit from having access to data on future jobs. This forward-looking information might help the model identify patterns or dependencies that would not be apparent if it only considered past and current jobs. In the intended application of the model, it is expected to predict the results of simulations for all jobs in the sequence, leveraging the full context of the simulation to make accurate predictions. For simplicity, this model is referred to as GRU further in this work.

The 'LSTM' model has a similar structure as the first model, but it uses the BiLSTM layers instead of the BiGRU layers [61, 39]. The LSTM model is described in Section 2.4.5, and the BiLSTM modification is described in Section 2.4.5. Capturing long-term dependencies in simulation results prediction can be beneficial because it allows the model to consider the influence of events over extended periods, leading to more accurate and informed predictions based on the historical context of the simulation data. Jobs that were submitted earlier and are still in execution can influence the jobs that are submitted subsequently. The motivation for employing BiLSTM layers instead of standard LSTM layers is the same as for choosing BiGRU over conventional GRU layers. This model is referred to as LSTM further in this work.

As the third model, we aim to utilize the Transformer model, adapted for numerical predictions [77]. The Transformer model is described in Section 2.4.7. We evaluated three types of Transformers for our task: original Encoder-Decoder Transformer, Encoder-Only Transformer and Decoder-Only Transformer. Encoder-Only Transformers have a simpler architecture compared to Encoder-Decoder models because they consist solely of the encoder part [35]. The main advantage of Encoder-Only Transformers in this context is their compatibility with the same training and evaluation procedures used for the GRU and LSTM models. In contrast to Encoder-Only Transformers, Decoder-Only, and Encoder-Decoder Transformers require direct access to the target data during training and a more complex autoregressive generation process during inference [7]. After experimenting with all three types of models, we have decided to limit our research to Encoder-Only Transformers. However, the applicability of Encoder-Decoder Transformers in this field presents an interesting research question for future work.

4.3. Sequence-to-Sequence Prediction

In Section 4.1, we discussed the data preparation process, particularly focusing on the transformation of simulations into a sequence of windows. Our models are designed to process a sequence of fixed-size windows, where each window of size n can be viewed as a matrix with n rows, each corresponding to a simulated job, and a column for each input parameter. For each input window, the model predicts an output window that retains the same number of rows, thereby maintaining a one-to-one relationship between input and output jobs, and includes a column for each output parameter. This approach ensures that for every row in the input window (representing one simulated job), exactly one output row is predicted, directly mapping the input sequence to the output sequence. Figure 4.4 illustrates this approach. If a sequence is split into overlapping windows, a single entry from the original sequence may appear in multiple input windows, resulting in potentially different predictions for that entry each time it is encountered. We address this issue in Section 4.5.

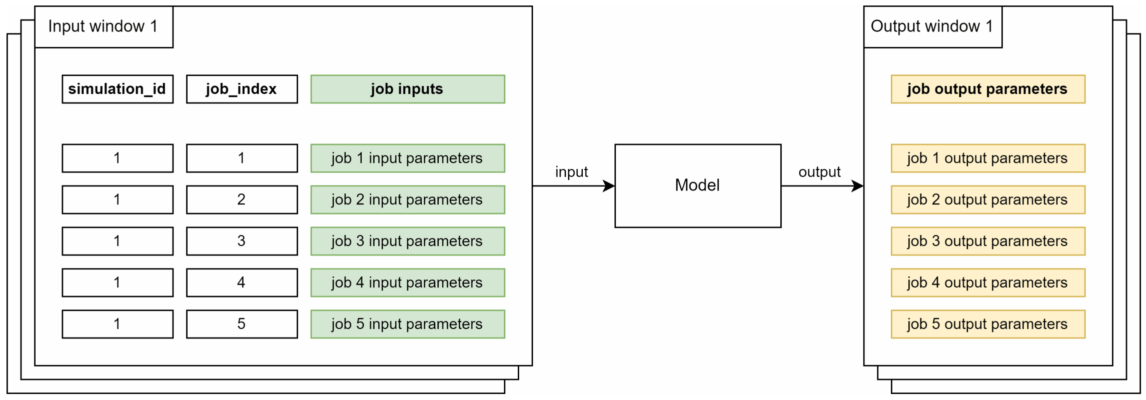


Figure 4.4.: Model generated an output window for each input window separately

4.4. Model Evaluation

During model evaluation, the predicted output parameters for each job are compared to the actual output parameters for that job. The goal of the evaluation is to compare the predictive capabilities of different models and select the best model architecture and hyperparameters in each scenario. In each scenario, the models are evaluated in two distinct tasks:

1. Interpolation: This task assesses the model's ability to generalize within the range of sequence lengths it was trained on. It tests the model's capacity to make accurate predictions for new sequences that are similar in length to those seen during training, demonstrating its ability to understand and apply learned patterns to slightly different but related scenarios.

2. Extrapolation: This task evaluates the model's ability to extend its predictions beyond the range of sequence lengths it encountered during training. It tests the model's capacity to infer and apply patterns to sequences that are longer or shorter than those it was trained on, demonstrating its ability to adapt to new situations and predict outcomes for sequences with lengths it has not previously seen.

In this research, we employ four metrics and two types of plots to evaluate our models. For each parameter, we utilize the regression metrics Mean Squared Error (MSE) [49], Root Mean Squared Error (RMSE) [19], Mean Absolute Error (MAE) [75], and Coefficient of determination (R-squared) [22] to assess the accuracy and reliability of our predictions. For the normalized data, MSE, RMSE, and MAE have a minimum value of 0, which indicates that the predicted values perfectly match the actual values. MSE of 1 indicates that the average squared deviation of the predictions from the actual values is equal to the variance of the normalized data. This suggests that, on average, the model's predictions are about one standard deviation away from the actual values. RMSE is the square root of MSE and provides a measure of the average magnitude of the prediction errors. An RMSE of 1 indicates that, on average, the model's predictions are one standard deviation away from the actual values, offering a more interpretable scale compared to MSE. MAE measures the average absolute deviation of the predictions from the actual values. An MAE of 1 suggests that, on average, the model's predictions deviate from the actual values by an amount equal to the mean absolute deviation of the normalized data. Unlike MSE and RMSE, MAE is not sensitive to the square of the errors, making it more robust to outliers. R-squared indicates the proportion of the variance in the dependent variable (predicted parameters) that is predictable from the independent variables (input parameters) in a regression model.

The value range of R-squared is $(-\infty, 1)$. A value of 1 indicates a perfect fit, and a value of 0 indicates that the model explains none of the variability of the response data around its mean. R-squared can be negative, indicating a model worse than a simple mean predictor. Table 4.1 depicts an example for an abstract scenario, which contains the accuracy metrics for five parameters. The model shows strong predictive capabilities for all parameters, predicting the 'compute_time' almost perfectly.

Table 4.1.: Accuracy metrics example

Parameter	**MSE**	**RMSE**	**MAE**	**R^2**
job_start	0.09730	0.31193	0.23544	0.90269
job_end	0.10032	0.31674	0.23737	0.89968
compute_time	0.00557	0.07463	0.04633	0.99443
input_files_transfer_time	0.10934	0.33067	0.23781	0.89066
output_files_transfer_time	0.01488	0.12196	0.04489	0.98512

In our analysis, we employ a specific type of visualization known as an accuracy plot to evaluate the predictive performance of our model [32]. This scatter plot positions the model's predictions on the y-axis against the actual values on the x-axis. Each point on the plot represents a single observation, with its location determined by the model's predicted value for that observation (y-coordinate) and the true value (x-coordinate). An ideal model, which predicts with perfect accuracy, would result in all points being positioned on the diagonal line where the actual values equal the predicted values, indicating a perfect match between predictions and reality. Deviations from this diagonal line illustrate discrepancies between predicted and actual values, providing a visual representation of the model's accuracy.

Although metrics are valuable for comparing different models, they primarily provide a high-level overview of performance. The accuracy plots, on the other hand, are instrumental in gaining deeper insights into model performance, revealing specific areas where improvements might be necessary. To enhance the interpretability of the accuracy plots, the predictions are scaled back to the scales of the original parameters. In Figure 4.5, the accuracy plot for the parameter 'job_start' is depicted for the same model whose metrics were presented in Table 4.1. On the plot, it is observable that in the interval $0.0 - 0.6 \times 10^7$ the model generates very accurate predictions. However, it systematically overestimates the predictions for the jobs with a 'job_start' value higher than 0.6×10^7. This plot gives us more information, than just the MSE value of 0.09730.

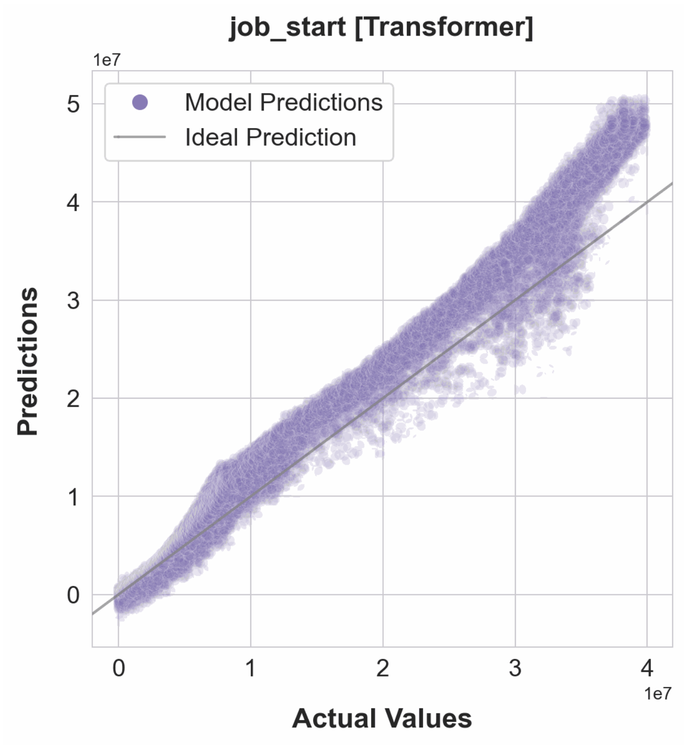


Figure 4.5.: Accuracy plot example

Additionally, we employ Kernel Density Estimation (KDE) plots in conjunction with accuracy plots [66]. KDE plots can reveal the density and distribution patterns of both the predicted and actual values, helping to identify biases, skewness, or outliers in the predictions. The x-axis contains normalized values of each parameter. The y-axis represents the density values, which indicate the probability of observing data points at different positions along the x-axis, effectively showing the distribution of the data. The area under the KDE curve should integrate to 1, meaning that the total probability across all possible values of the variable is 1.

By comparing the KDE plots of the predicted versus actual values, it is possible to identify systematic errors in the model's predictions. There are two typical signs of error in KDE plots: underestimation and overestimation. Underestimation occurs when the KDE plot consistently lies below the actual data distribution, suggesting that the model is systematically predicting lower values than observed. Overestimation is observed when the KDE plot consistently lies above the actual data distribution, indicating that the model is systematically predicting higher values than observed. For instance, if the model consistently overestimates or underestimates values within a specific range, this pattern will be evident in the KDE plot. A KDE plot for 'job_start' is presented in Figure 4.6, created using the same data as the accuracy plot in Figure 4.5. While the accuracy plot provides a good overview of the model's performance, the KDE plot distinctly highlights overestimations and underestimations within specific parts of the distribution, offering a more nuanced understanding of the model's predictive behavior.

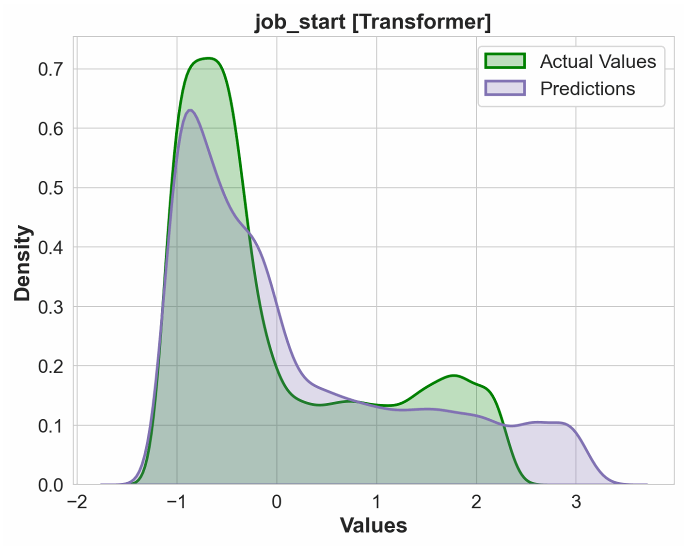


Figure 4.6.: KDE plot with overestimation for greater parameter values jobs and underestimation for smaller parameter values

However, KDE plots do not always offer more informative insights than accuracy plots. In Figure 4.7, both accuracy and KDE plots are presented for the parameter 'compute_time'. Despite the model's near-perfect capture of the distribution as shown in the KDE plot, the accuracy plot reveals a splitting of points into two distinct beams. This behavior can be caused by differences in the computational capabilities of two node groups, which were used as a platform in the simulation. This pattern suggests a systematic error, highlighting an aspect of model performance that might not be as apparent from the KDE plot alone.

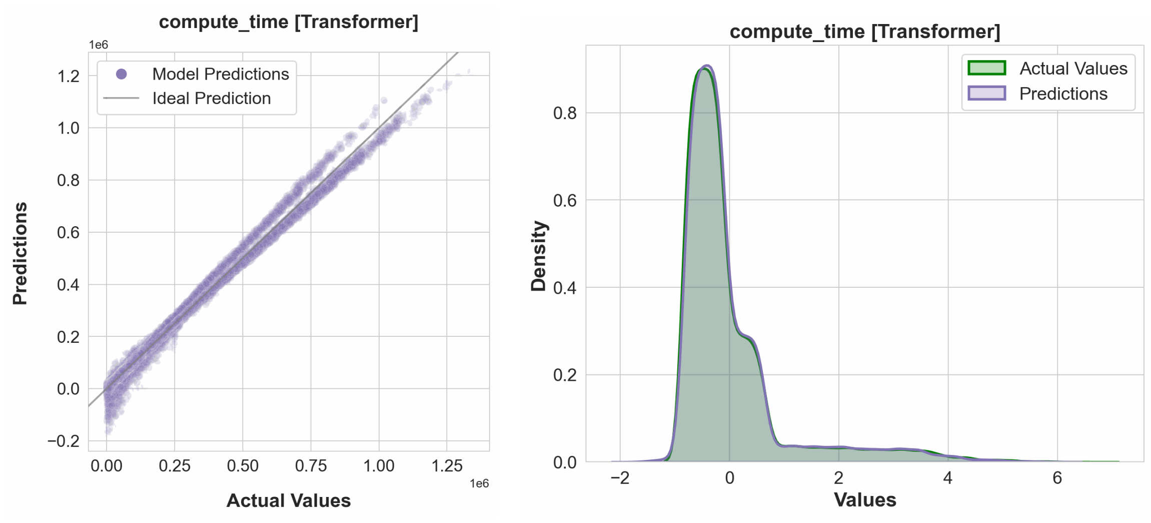


Figure 4.7.: Accuracy plot and KDE plot for the same parameter

4.5. Data Reconstruction

An essential aspect of our evaluation involves the reconstruction of data after prediction, to ensure the integrity and usability of model outputs for further analysis and interpretation. This process is especially crucial when employing varying window overlaps and padding, as these techniques introduce duplicates into the data, potentially impacting the overall data distribution and the accuracy metrics. To remove duplicates from the data, the following steps can be followed:

1. Transform the input and output sequences of windows into a plain array.
2. Use the input array of jobs to identify all indexes of duplicates that have the same 'simulation_id' and 'job_index' combinations, except for the first occurrence.
3. Remove all jobs from the input and output arrays that correspond to these indexes.

This approach ensures that no duplicates remain in the data and only one padding entry is retained, which should not significantly affect the metrics and distributions. Figure 4.8 illustrates this approach.

Figure 4.9 presents two KDE plots of the same model, trained with a window size of 200 and an overlap size of 100. The left plot displays the KDE plot of the data before removing duplicates introduced by windowing, while the right plot shows the KDE plot of the data after the removal of duplicates. The data used for model evaluation contains a job batch which consists of job with identical resource demands, which were assigned simultaneously on startup. The total batch size if five times greater the number of job slots in the system. The distribution of job staring time in five stages groups is evident in the right plot but cannot be discerned in the left plot.

4.6. Hyperparameters

The choice of optimal hyperparameters is an important part of machine learning, as it significantly influences the performance and effectiveness of the models [79].

One of the most important hyperparameters is the learning rate [68]. After evaluating various learning rates, we opted for a fixed learning rate of 0.001 for all models. Additionally, we employ a learning rate scheduler that reduces the learning rate by a factor of 0.1 when the change in the average loss over the last 5 epochs falls below the threshold of 0.0001. These parameters provided the best balance between convergence speed and stability, leading to optimal training performance across our range of models.

As for the loss function, the MSE Loss was chosen after evaluating several loss functions [36]. The other evaluated loss functions were: MAE Loss (L1Loss) [59], Huber Loss (SmoothL1Loss) [3]. Huber Loss is a hybrid loss function that behaves like MSE for small errors and like MAE for large errors, making it less sensitive to outliers than mean squared error. MAE loss may be less sensitive to reducing outliers in the model's predictions, and

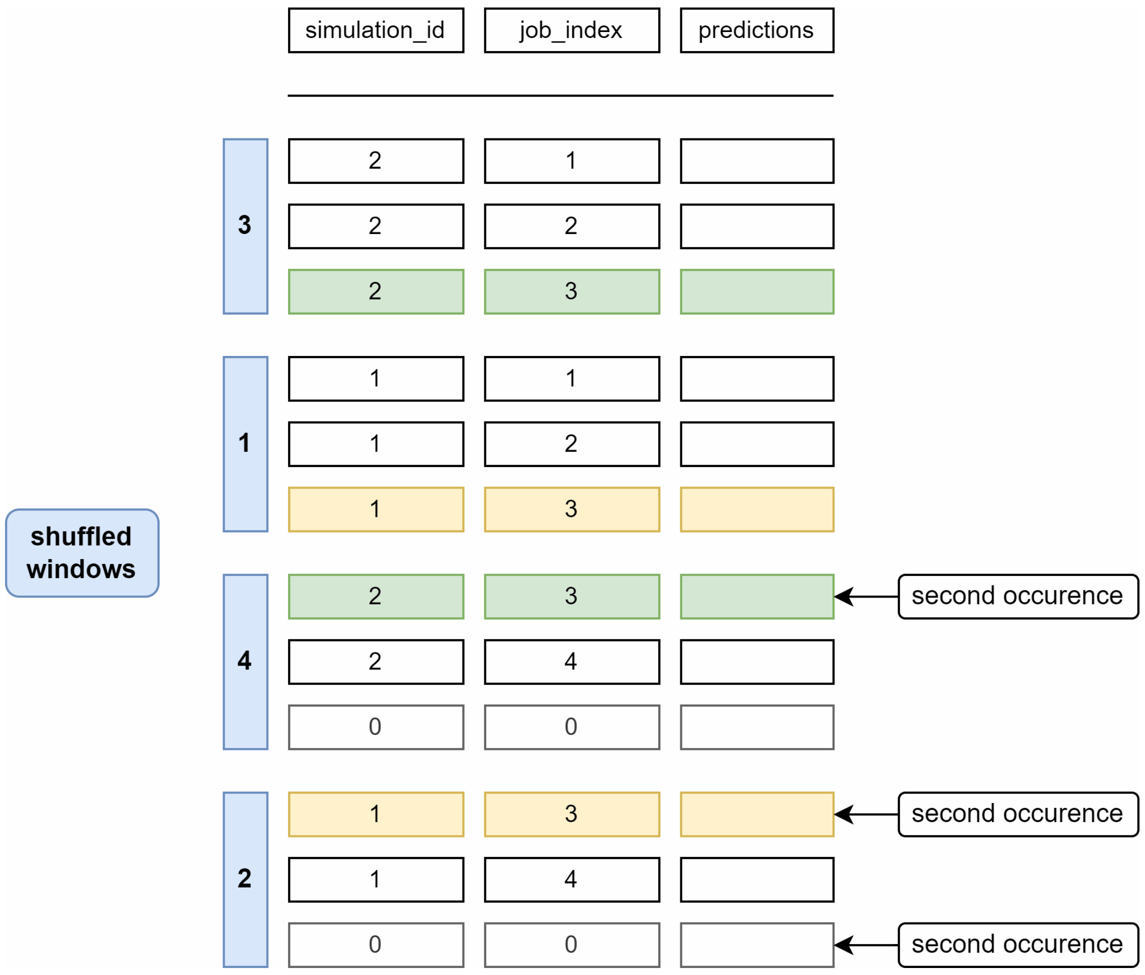


Figure 4.8.: Use simulation_id and job_index as primary key to identify duplicates

Huber loss performed similarly to MSE but required approximately twice as much time for each epoch. This time difference can be explained by the use of a conditional decision within the calculation of Huber loss.

The choice of MSE Loss aligns with our intention to penalize larger errors more heavily than smaller errors, reflecting our preference for a model that emphasizes the importance of minimizing significant discrepancies between predicted and actual values.

The free hyperparameters are:

- **Hidden Size**: Determines the dimensionality of each hidden layer.
- **Window Size**: Specifies the number of consecutive data points considered in each input window.
- **Window Overlap**: Defines how much consecutive windows overlap.
- **Batch Size**: Indicates the number of samples processed before the model's internal parameters are updated.

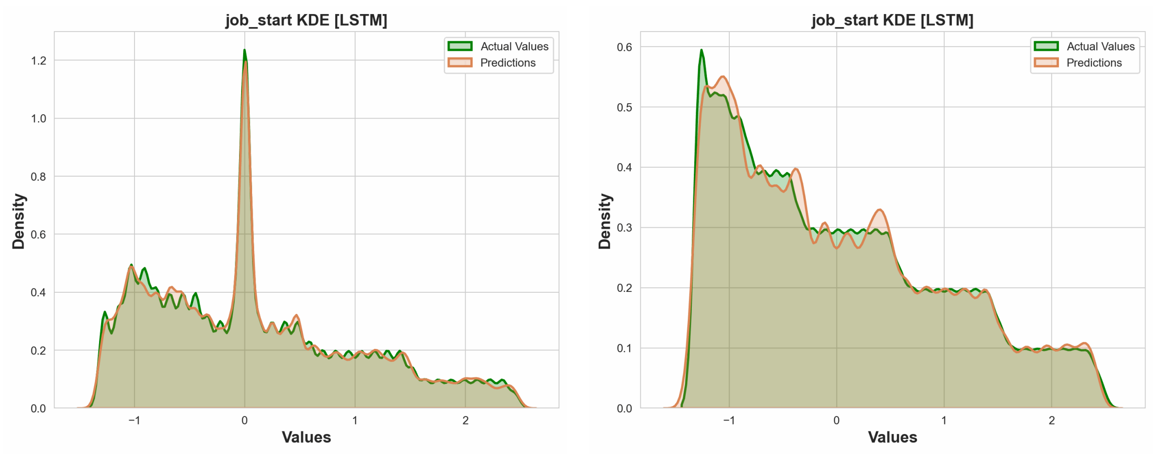


Figure 4.9.: KDE plot with uncleared data (left) and with cleared data (right)

- **Layers**: Determines the number of hidden layers (BiGRU, BiLSTM and Encoder layers) in the model.
- **Heads**: the number of attention heads (only for Transformer)

There are many strategies for optimizing hyperparameters in machine learning models. Given the extensive search space, employing a grid search would be highly resource- and time-consuming [44]. Alternatively, various hyperparameter tuning algorithms exist, such as the Tree-structured Parzen Estimator [73]. These automated algorithms typically aim to find a configuration that minimizes the training loss or optimizes metrics, such as MSE. However, as discussed in Section 4.4, in this use case, metrics alone do not provide a complete assessment, as the primary goal of the model is to match the actual data distribution. Therefore, to determine which model performs better, the KDE plots and the accuracy plots are empirically evaluated. In the context of our work, this process cannot be feasibly automated due to the necessity of evaluating various types of visualizations.

We optimize the hyperparameters sequentially, iterate several times, and use random start configurations to reduce the risk of sticking in the local optima. This approach for hyperparameter optimization is presented in Figure 4.10. At first, we train 10 models with random parameters with predefined ranges. After the initial training phase, we evaluate the performance of each model using KDE plots, accuracy plots, and accuracy metrics. We then select the best 3 models from this initial group. For each of these models, we modify one free hyperparameter at a time, train, and evaluate the models with different variations of this parameter. After determining the optimal value for a parameter, we fix it, retain the best-performing model from the last iteration, discard the others, and proceed to adjust the next parameter. This process involves training new models, evaluating them, and comparing the outcomes. We repeat this cycle until we have iterated through all the parameters once. In the final step, we compare the three resulting models and select the best one. The parameters are adjusted in the following order: Hidden Size, Window Size, Window Overlap, Layers, Heads, Batch Size. This order was established after training and evaluating the models for the different scenarios, and it produced better results than other orders that were tested.

To compare the models, we needed to determine the order of importance of different plots and metrics for each output parameter. As a result, we establish the following list of priorities:

1. If the model fails to match the distribution for sequences with lengths seen during model training (interpolation), it is not subjected to further evaluation. Underestimations and overestimations are not accepted.

2. Matching the distribution for sequences with lengths unseen during model training (extrapolation) is the most important criterion. There is a high penalty for underestimations and overestimations. Matching 'job_start', 'job_end', and 'compute_time' parameters are considered more important than other output parameters, such as file transfer time.

3. Achieving good metrics values in extrapolation tasks. The most crucial metrics are R-squared and MSE, with metrics for 'job_start', 'job_end', and 'compute_time' being the most significant.

4. If there are no visible differences in KDE and accuracy plots, the metrics are used for comparison. Only differences in metrics larger than 0.01 are considered significant for analysis. If the difference in one metric is not significant, preference is given to the simpler model, characterized by a smaller hidden size, fewer layers, smaller window overlap, and larger batch size.

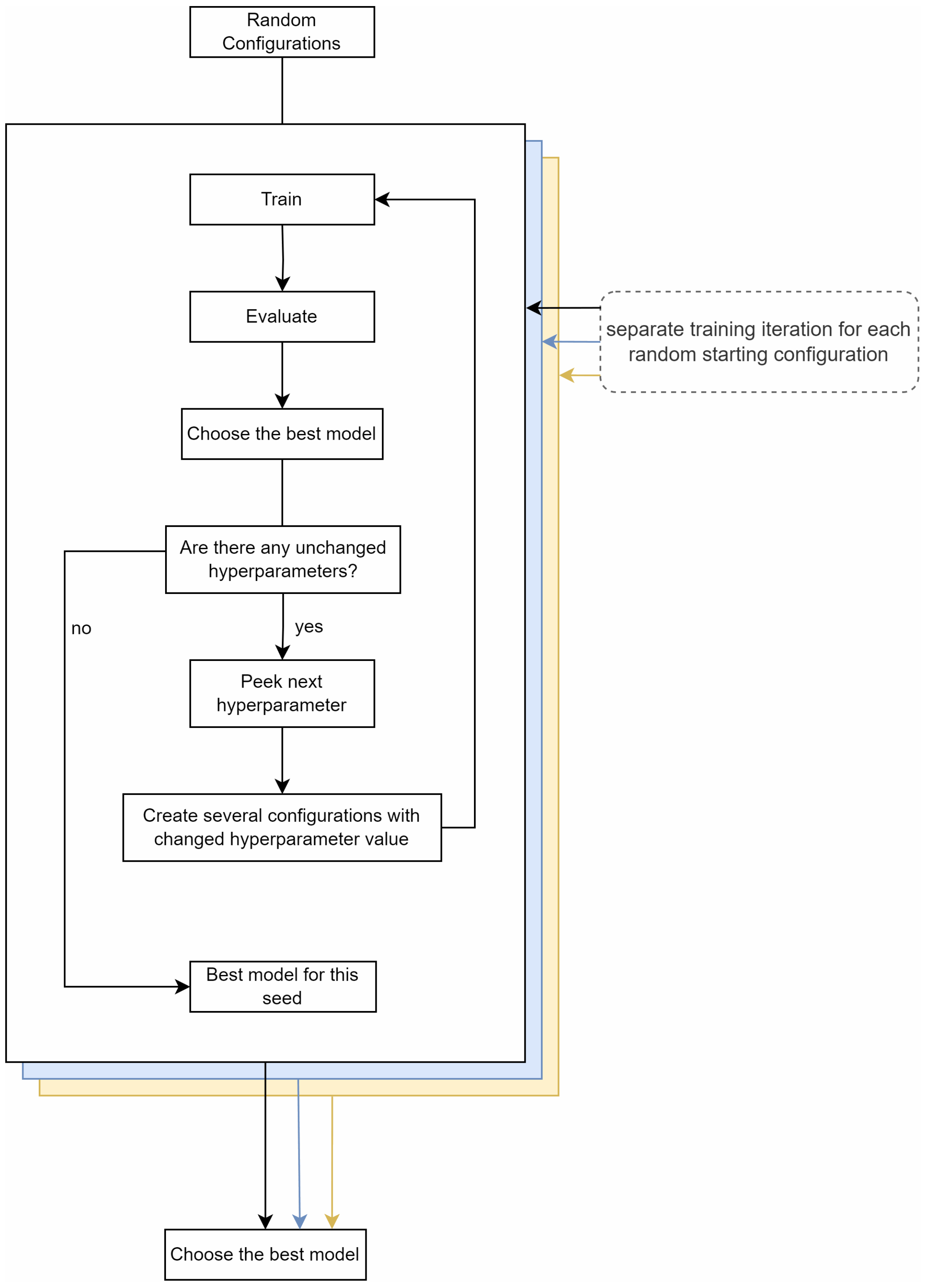

Random Configurations
Train
Evaluate
Choose the best model
Are there any unchanged hyperparameters?
no
yes
Peek next hyperparameter
Create several configurations with changed hyperparameter value
Best model for this seed
separate training iteration for each random starting configuration
Choose the best model

Figure 4.10.: Sequential hyperparameter optimization with random starting configurations

5. Experimental Design

Our experiment is structured into two distinct parts. In the first part, we focus on refining and evaluating our sequence-to-sequence regression concept, utilizing exclusively the jobs from the DCSim output. This first part encompasses three scenarios, each with increasing complexity.

The first scenario detailed in Section 5.2 serves as a proof of concept and provides a foundation for the validation of our design decisions. We focus on jobs originating from a single job distribution, and operating on a fixed platform with minimal complexity. The DCSim configurations are described in Section 2.2.1. The second scenario detailed in Section 5.3 introduces complexity by predicting simulation results for jobs from varied job distributions. The underlying platform is the same as in the first scenario. The third scenario detailed in Section 5.4 focuses on predictions for heterogeneous jobs, similar to the second phase, simulated on a more complex platform. This scenario results in models, that can handle heterogeneous jobs, but should be trained for each platform. Table 5.1 summarizes these scenarios.

Scenario	Jobs	Platform Complexity
I	homogeneous	Sgbatch
II	heterogeneous	Sgbatch
III	heterogeneous	GridKa

Table 5.1.: Three scenarios with increasing data and platform complexity

The second part of our experiment is designed to incorporate auxiliary information about platform topology and is described in Section 5.5. In this part, we assess the model's capacity to generalize across various platforms. To achieve this, we conduct simulations on different platforms and input the platform data into the model alongside the jobs data.

5.1. Tasks Description

The simulations were performed and divided into two datasets, following a 70/30 split. The training dataset, comprising 70% of the simulations, was utilized for training the models. The evaluation dataset, consisting of the remaining 30%, included only unseen simulations and was used for the interpolation task. The simulations in the evaluation dataset had lengths represented in the training dataset.

For the extrapolation task, we utilize the same models that were employed for the interpolation task. The models are applied to a dataset comprising simulations, each containing five times more jobs than the largest simulation in the training dataset. These jobs are stored in a separate extrapolation dataset, which is not used during the model training.

5.2. First Scenario: Homogeneous Jobs

For the first scenario of our research, we execute a total of 10.000 simulations. These were organized into 10 batches, each containing 1.000 simulations. The simulations were structured to include varying numbers of jobs, with batches for 1, 10, 20, 50, 100, 250, 500, 1.000, 1.500, and 2.000 jobs respectively. This approach allowed us to analyze the system's behavior across a spectrum of workload sizes.

From the data generated in this scenario, we create two distinct datasets. The first dataset incorporated all 10.000 simulations and was dedicated to the final training and evaluation of the interpolation capabilities of our model. For the extrapolation task, we conduct extra set of 10 simulations, each containing 10.000 jobs. Model training and evaluation results are described in Chapter 7.

Listings 5.1 and 5.2 present an example workload and dataset configuration, used in simulations. All job resource demands are constant, only the 'num_jobs' parameter varies between workload configurations.

The platform configuration used for these simulations contains three worker nodes with 60 CPU cores in total and one scheduler node. All nodes in the network are interconnected following a star topology, with a central gateway serving as the primary hub. Each link connecting the nodes to the gateway possesses uniform bandwidth and latency. The datasets required for processing by the jobs are not stored directly on the worker nodes. Instead, they are located on a separate storage server referred to as 'RemoteStorage' in dataset configuration, which is also connected to the central gateway. Full platform configuration is presented in Listing A.1.

In Section 4.3 we described an approach for the sequence-to-sequence predictions, which incorporates input and output windows. The parameters 'index', 'flops', 'input_files_size', and 'output_files_size' are used in input windows and the parameters 'job_start', 'job_end', 'compute_time', 'input_files_transfer_time', 'output_files_transfer_time' are used om the output windows. The output parameters are the parts of the DCSim output file, which is presented in Section 2.2.2. Input and output parameters are described in Section 2.2.3. All parameters contain floating point numbers with 32-bit precision. The value of 'submission_time' of all jobs in this scenario is identical and therefore is not used as the input parameter.

Listing 5.1: Workload configuration with uniform jobs, only the 'num_jobs' is changed

```
{
    "simple_uniform": {
        "num_jobs": 50,    <!-- 1, 10, 50, 100, 250, 1.000, 1.500, 2.000 -->
        "infiles_per_job": 10,
        "cores": {
            "type": "histogram",
            "counts": [0,1]
        },
        "flops": {
            "type": "gaussian",
            "average": 2886000000000,
            "sigma": 0
        },
        "memory": {
            "type": "gaussian",
            "average": 2000000000,
            "sigma": 0
        },
        "outfilesize": {
            "type": "gaussian",
            "average": 16000000,
            "sigma": 0
        },
        "workload_type": "streaming",
        "submission_time": 0,
        "infile_datasets": "simple_uniform"
    }
}
```

Listing 5.2: Dataset configuration with uniform files

```
{
  "simple_uniform": {
    "location": "RemoteStorage",
    "num_files": 1000000,
    "filesize": {
      "type": "gaussian",
      "average": 500000000,
      "sigma": 0
    }
  }
}
```

5.3. Second Scenario: Heterogeneous Jobs

For the second scenario, we perform 7.000 simulations organized into 7 batches, each containing 1.000 simulations. The simulations are structured to include varying numbers of jobs, with batches for 10, 20, 50, 100, 250, 500 and 1.000 jobs respectively. Additionally, we perform 10 simulations each containing 5.000 jobs for the extrapolation task. Model training and evaluation results are described in Chapter 8.

Listing 5.3 presents a part of workload configuration with two job classes: 'Analysis_T1' and 'Digi_T1'. The full workload configuration is presented in Listing A.3 and contains five different job classes, each job contains its own distributions for all resource demands. Listing 5.4 presents the corresponding part of dataset configuration. Full dataset configuration is presented in Listing A.5. The platform remains the same as in the first scenario.

Listing 5.3: Part of the workload configuration with several workload classes

```
{
    "Analysis_T1": {
        "num_jobs": 15,
        "cores": { <!-- distribution, instead of fixed value -->
            "type": "histogram",
            "counts": [0, 206746, 35, 0, 3709, 0, 0, 0, 406],
            "bins": [-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5]
        },
        "flops": {
            "type": "histogram",
            "counts": [210623, 263, 4, 6, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [650, 204277, 3161, 1560, 20, 0, 949, 0, 225, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type": "histogram",
            "counts": [210319, 577, 0, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10, <!-- set for each workload class -->
        "infile_datasets": "Analysis_T1"
    },
    "Digi_T1": {
        "num_jobs": 15,
        "cores": {
            "type": "histogram",
            "counts": [0, 30704, 101899, 0, 86729, 0, 0, 0, 626]
        },
        "flops": {
            "type": "histogram",
            "counts": [74119, 141559, 3664, 38, 213, 317, 40, 8, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [131371, 8158, 54096, 20739, 192, 0, 5398, 0, 4, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type": "histogram",
            "counts": [155865, 59084, 1274, 593, 280, 211, 383, 131, 1528, 255],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 20,
        "infile_datasets": "Digi_T1"
    }
}
```

In the second scenario, we use five workload classes with different 'submission_time' values. In models for this scenario, the 'submission_time' is an input parameter. Other input and output parameters and dataset organization are the same as in the first scenario.

5.4. Third Scenario: GridKA Platform

For the third scenario, we perform 10.000 simulations organized into 10 batches, each containing 1.000 simulations. The simulations are structured to include varying numbers of jobs, with batches for 5, 10, 50, 100, 250, 500, 1.000, 1.500, and 2.000 jobs respectively. Additionally, we perform 100 simulations containing 10.000 jobs for the extrapolation task. Model training and evaluation results are described in Chapter 9.

Listing A.4 presents a workload configuration, Listing A.6 presents a dataset configuration, and Listing A.2 presents a platform configuration. Dataset and workload configurations are similar to configurations, used in the second scenario. The new platform configuration describes an interconnected network of two data centers, GridKa and DESY. GridKa features a cluster comprising 10 worker nodes, each with identical characteristics. In contrast, Listing 5.4: Part of the dataset configuration with several datasets

```json
{
  "Analysis_T1":{
    "location": "RemoteStorage",
    "num_files": 1000,
    "filesize": {
        "type": "histogram",
        "counts": [208159, 967, 701, 447, 270, 154, 72, 31, 38, 6],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  },
  "Digi_T1":{
    "location": "RemoteStorage",
    "num_files": 1000,
    "filesize": {
        "type": "histogram",
        "counts": [157564, 3767, 32275, 353, 25151, 89, 0, 0, 759, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  }
}
```

DESY is equipped with a single worker node, which boasts greater power compared to a single GridKa node. Both data centers have dedicated cache nodes. In this scenario, all datasets are stored in the GridKa cache node. The GridKa cluster possesses a total of 420 CPU cores, while the DESY worker node is equipped with 200 CPU cores.

Model input and output parameters are the same as in the second scenario. Workload classes used in this scenario are based on the real workload in Tier-1 and Tier-2 data centers.

5.5. Fourth Scenario: Platform Generalization

In prior scenarios, models were trained and evaluated for specific platforms and demonstrated generalizability across various workloads. In this scenario, we aim to add the platform information to the model inputs and outputs. To achieve this, we construct a representation of the topology graph of the simulated platform, which is then passed into our models together with the simulation data. The outcomes of this experiment are described in Chapter 10.

5.5.1. Platform Information

In earlier scenarios, our approach involved processing a sequence of individual jobs. For each job, the models predicted several parameters: 'job_start', 'job_end', 'compute_time', 'input_files_transfer_time', and 'output_files_transfer_time'. These predictions were based on a set of input parameters for each job, specifically 'index', 'submission_time', 'flops', 'input_files_size', and 'output_files_size'. In this scenario, we pass a platform topology in the form of two auxiliary sequences: (i) a nodes sequence and (ii) a links sequence. These auxiliary sequences provide contextual information or supplementary data that assists the model in making more informed predictions for each job in the primary sequence.

The nodes sequence contains one object for each node in the platform configuration. Each node represents a computing or infrastructure element in the platform, such as a cluster, worker, scheduler, storage, cache or router. Each node by id, type, cores, number of flops per second of each core, RAM, disc capacity, disk write and read bandwidths. Additionally, for each node is explicitly described, which cluster is this node part of. Cluster id 0 means, that this node is not a part of any cluster. The auxiliary nodes sequence for the platform from Listing A.2 is presented in Listing 5.5.

The links sequence contains one object for each link in the platform configuration. Each link is defined by id, source node id, destination node id, link latency, link bandwidth, and whether a fatpipe link sharing policy is enabled for this link. Node id, used in the links sequence, is a reference to the node in the nodes sequence. The auxiliary links sequence for the platform from Listing A.2 is presented in Listing 5.6.

Listing 5.5: Auxiliary nodes sequence with GridKa and DESY nodes

```
node_name;node_index;node_type;node_type_index;speed_mf;cores;ram_gib;disk_tib;
    disk_read_bw_mbps;disk_write_bw_mbps;in_cluster
Tier1;1;cluster;0;0;0;0;0;0;0;0
Tier10;2;worker;1;2555;42;1187;0;0;0;1
Tier11;3;worker;1;2555;42;1187;0;0;0;1
Tier12;4;worker;1;2555;42;1187;0;0;0;1
Tier13;5;worker;1;2555;42;1187;0;0;0;1
Tier14;6;worker;1;2555;42;1187;0;0;0;1
Tier15;7;worker;1;2555;42;1187;0;0;0;1
Tier16;8;worker;1;2555;42;1187;0;0;0;1
Tier17;9;worker;1;2555;42;1187;0;0;0;1
Tier18;10;worker;1;2555;42;1187;0;0;0;1
Tier19;11;worker;1;2555;42;1187;0;0;0;1
Tier2;12;worker;1;2209;200;500;0;0;0;0
GridKA_dCache;13;storage;2;1000000;10;0;7000;920;920;0
WMSHost;14;scheduler;3;10000;10;16;0;0;0;0
GridKAgateway;15;router;4;0;0;0;0;0;0;0
KITgateway;16;router;4;0;0;0;0;0;0;0
DESY_dCache;17;cache;5;1000000;10;0;7000;920;920;0
DESYGridgateway;18;router;4;0;0;0;0;0;0;0
DESYgateway;19;router;4;0;0;0;0;0;0;0
```

Listing 5.6: Auxiliary links sequence with GridKa and DESY links

```
link_name;link_index;src_node_name;src_node_index;dst_name;dst_node_index;
    bandwidth_mbps;latency_us;is_fatpipe
GridKA_sched;1;GridKAgateway;15;WMSHost;14;115;0;0
GridKA_Tier1_FATPIPE;2;GridKAgateway;15;Tier1;1;1150;0;1
GridKA_Tier1;3;GridKAgateway;15;Tier1;1;2300;0;0
GridKA_dcachepool_FATPIPE;4;GridKAgateway;15;GridKA_dCache;13;460;0;1
GridKA_dcachepool;5;GridKAgateway;15;GridKA_dCache;13;920;0;0
GridKA_to_KIT;6;GridKAgateway;15;KITgateway;16;1150;0;0
KIT_to_DESY;7;KITgateway;16;DESYgateway;19;115;0;0
DESY_Tier2;8;DESYGridgateway;18;Tier2;12;460;0;0
DESY_dCachepool;9;DESYGridgateway;18;DESY_dCache;17;460;0;0
DESYGrid_to_DESY;10;DESYGridgateway;18;DESYgateway;19;1150;0;0
```

5.5.2. New Job Features

We are introducing modifications to the job parameters in the primary job sequence. Along with the existing input and output parameters, each job entry includes two new parameters.

The first parameter is 'dataset_node_index'. It contains the index of the node where the dataset required by a specific job is located. In scenarios where the dataset is located on the node 'GridKA_dCache', the 'dataset_node_index' would be set to 13, as per the node sequence presented in Listing 5.5. This additional parameter aims to provide more contextual information to the model, specifically regarding the location of the dataset within the network topology. We make a simplifying assumption that the dataset required for a specific job is stored on a single node, and the dataset required for a specific job is not distributed or scattered across multiple nodes within the network topology. This parameter is an input parameter and is used to enhance the predictions.

The second new parameter is 'machine_index', which is an output parameter. This parameter contains the index of the worker node where the job is executed. For instance, if a job is processed on the DESY worker node named 'Tier2', the 'machine_index' parameter would be assigned the value 12. This corresponds to the order of nodes as detailed in Listing 5.5. It enables our model to predict, on which node the job will be executed.

5.5.3. Incorporation of Platform Information

We need a way to incorporate the platform information into the prediction process. Complete platform data is included in the model with each input window. New input columns are created to encapsulate information about nodes and links. The window size must be equal to or exceed the number of nodes and links. Otherwise, the platform information can not be fitted in the window. The shortest subsequence is padded with zeros to match the window size.

This concept is illustrated in Figure 5.1. This figure presents a sequence of 10 jobs, contained in two windows without overlap. Node information (e.g. node index, number of CPUs, computational speed of a single CPU, size of RAM and disk) and link information (e.g. indexes of source and destination nodes, link bandwidth, and latency) are added to each window. In this example, each window contains the parameters of three nodes and two links, which are passed as input along with the job parameters.

An alternative approach could be the simultaneous processing of three separate sequences: a sequence of jobs (as in all previous scenarios), a sequence of nodes information (as in Listing 5.5), and a sequence with links information (as in Listing 5.6). The LSTM and GRU models can process only one sequence in one model, but Transformer models do not have this limitation. Investigating how Transformers handle multiple separate sequences was not performed in this work due to time constraints; however, it remains an interesting topic for future research.

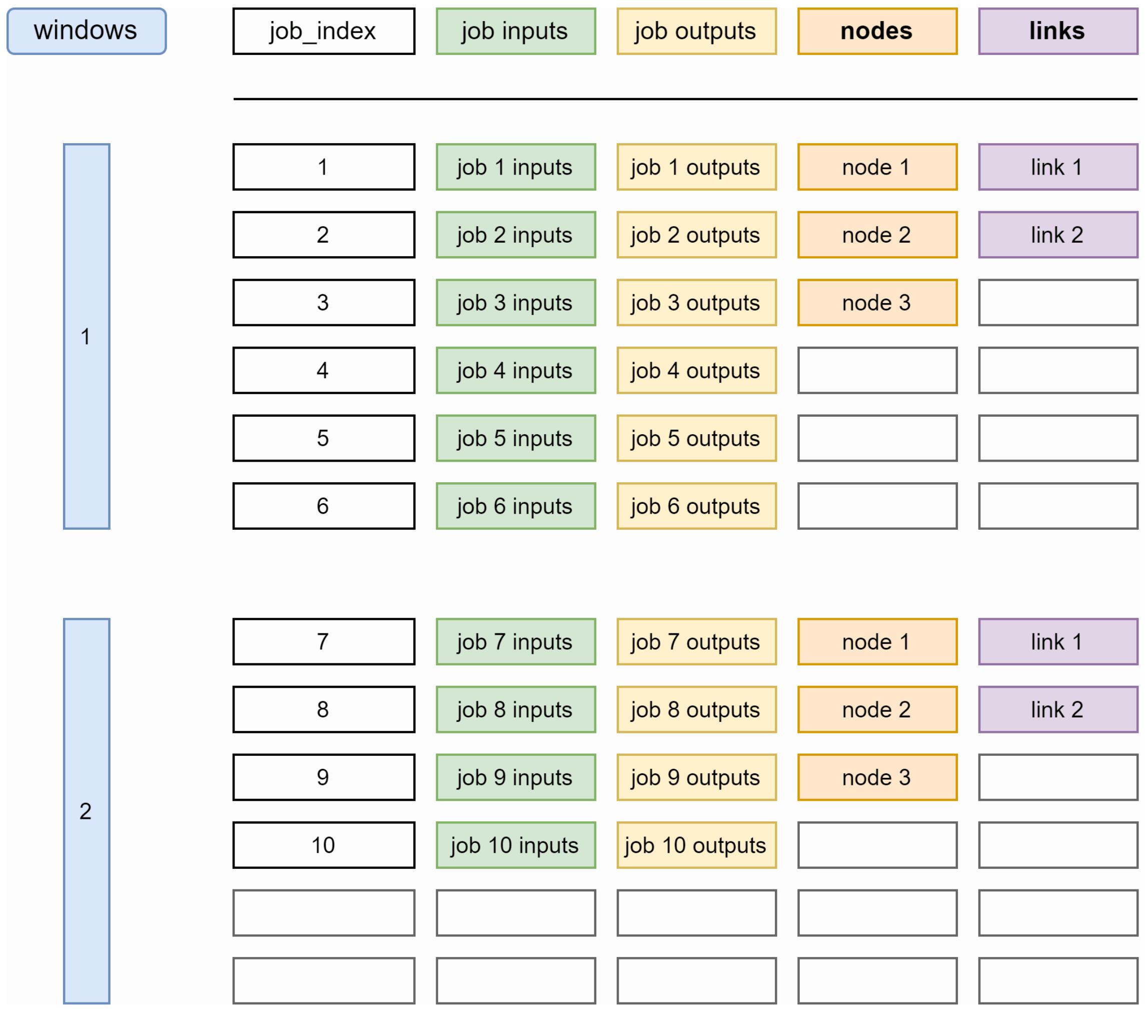


Figure 5.1.: Information about node and links in each window

5.5.4. Training Data

We create 15 different platform configurations and performed simulations using them. An example of a platform configuration is presented in Listing 2.3. For each platform configuration, we perform the simulations with 5, 10, 20, 50, 100, 250, 500, 1.000, 1.500, and 2.000 jobs, with 10 simulations for each sequence length. We perform 100 simulations for each platform configuration, 1.500 simulations in total. Additionally, we reuse the 10.000 simulations from the third scenario.

These 15 configurations describe the modifications of Sgbatch and GridKa platforms, used in previous scenarios. In each modification, we change the value of one parameter (e.g. number of CPU cores in different worker nodes), the values of several parameters (e.g. link bandwidth and latency), or the topology (e.g. new nodes, clusters, or links). The full list of base platforms and their modifications is presented in Table 5.2. For each configuration, we save the files with descriptions of nodes and links in the format presented in Listings 5.5 and 5.6 with the simulations data.

Base Platform	Modification
Sgbatch	Increased latency in all links
Sgbatch	Decreased bandwidth and increased latency in all links
Sgbatch	Increased number of CPU cores in all nodes
Sgbatch	Decreased number of CPU cores, increased RAM volume in all nodes
Sgbatch	Decreased disk bandwidth in all nodes
Sgbatch	Removed one worker node and corresponding links
GridKa	Increased bandwidth in all links
GridKa	Decreased number of CPU cores in all nodes
GridKa	Cache containing the dataset moved from GridKa to DESY
GridKa	Cache in DESY, increased core speed in all nodes
GridKa	Cache in DESY, decreased all resources in GridKa nodes
GridKa	Cache in DESY, increased number of CPU cores in DESY nodes
GridKa	Cache in DESY, more worker nodes in DESY
GridKa	Cache in DESY, less worker nodes in GridKa
GridKa	More worker nodes in DESY, less worker nodes in GridKa

Table 5.2.: Platform Modifications used for Model Training

5.5.5. Training Process

Training of models for this scenario occurs in two stages. At first, the model is trained similarly to the model from the third scenario, described in Section 5.4, using the same workload and platform configurations. Then, the initial learning rate is lowered from 0.001 to 0.0001 to prevent the weights learned from the base platform from being overwritten by training on new platforms.

After this, we iterate through the 15 training datasets which contain the simulation data from different platforms. We perform 10 iterations over all datasets, the model is trained in each iteration on each dataset for 100 epochs.

5.5.6. Task Description

After training, each model is evaluated on three platforms with increasing complexity. The first platform is one platform from 15 platforms, used during model training. The second platform represents the modification of node parameters with more CPU cores, more RAM, and more SSD storage. This modification is not present in the training dataset, the platform topology is the same as in the previous platform. The third platform represents the modification of both nodes and links parameters and platform topology. New worker nodes and corresponding links are added to GridKA and DESY zones.

The goals of these three tasks are: (i) to assess how well the model trained on multiple platforms performs compared to a model trained specifically for one platform; (ii) to evaluate the generalizability of the models when the platform parameter values change; and (iii) to assess the generalizability of the models when the platform's topology changes.

6. Infrastructure and Implementation

This chapter outlines the infrastructure and methodologies employed to generate simulation data, store it in a database, and create datasets for model training and evaluation.

6.1. Data Generation

The research presented in this thesis necessitated the generation of a substantial volume of simulation data, specifically utilizing DCSim. To efficiently manage and expedite this process, we employ the BWUniCluster. This facility improved our capability to run the simulations in parallel, thereby enhancing the overall scalability of data generation.

Key to our methodological approach was the development of a suite of tools designed to automate various aspects of the simulation process. At first, two tools for automated simulation configuration and scheduling were implemented: `start-simulations.py` and `schedule-simulations.py`[1]. With the implementation of these tools, we automated the data generation process, which enabled us to initiate the desired number of simulations.

6.2. Data Storage

Upon completion of the simulations, the results are ready to be transferred to a database for further usage. However, not all simulations proceed as expected. In certain extreme cases, simulations may encounter issues such as running out of memory or being aborted due to exceeding the allocated execution time limit. It is essential to identify and exclude these problematic simulations from being saved into the database, as they do not provide accurate or complete data and could potentially skew the results and analysis. For this purpose, a third tool `extract-simulations.py` was implemented. The database is not deployed on the BWUniCluster but is set up locally. The database backups are created automatically and stored in the cloud, which significantly minimizes the risks associated with data loss due to hardware failure[2].

[1]All scripts and models used in this work are available at `https://gitlab.kit.edu/kit/kastel/sdq/stud/abschlussarbeiten/masterarbeiten/valerii-zhyla`

[2]The replication dataset is available at `https://zenodo.org/records/10977016`

7. First Scenario: Homogeneous Jobs

This chapter describes the data preparation and model training processes specifically tailored for predicting outcomes of DCSim simulations with the simplest workload and dataset configuration, where all jobs have uniform resource demands. The data preparation process for this scenario is described in Section 5.2.

7.1. GRU

We follow the hyperparameter optimization described in Section 4.6 to find the best model parameters for this specific dataset. A total of 32 GRU models were trained. The best GRU model used one BiGRU layer with a hidden size of 128, a window size of 50, no overlap, and a batch size of 128.

7.1.1. Interpolation

Figures 7.1 and 7.2, and Table 7.1 present the results achieved by the GRU model in the interpolation task. As evident from the table and plots, the GRU model predicts all parameters except 'compute_time' with very high accuracy in this task. The distributions of actual values and predictions in Figure 7.2 are almost identical. The accuracy plot in Figure 7.1 for 'compute_time' reveals anomalies – the entire dataset comprises only two distinct 'compute_time' values. This could be attributed to the uniformity of the 'flops' parameter values across all jobs in this task, coupled with the similarity in computational resources of two out of the three nodes in the platform. All three models used in this work cannot precisely predict a single number if it was present in the dataset; variations in weights and biases will introduce some spread in the predictions. Due to the uniformity of all jobs in the sequence, all sequences exhibit a high degree of similarity.

7.1.2. Extrapolation

Figures 7.3 and 7.4, and Table 7.2 present the results achieved by the GRU model in the extrapolation task. In this new scenario, the GRU model encountered an unseen task and demonstrated a notable success in predicting 'job_start' and 'job_end' with high precision, even accurately capturing the distribution of these parameters to a large extent. However, the model completely failed in predicting 'compute_time', 'input_files_transfer_time',

Table 7.1.: GRU: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00035	0.01879	0.01405	0.99965
job_end	0.00036	0.01889	0.01413	0.99964
compute_time	0.10867	0.32964	0.16123	0.89131
input_files_transfer_time	0.00179	0.04233	0.02867	0.99821
output_files_transfer_time	0.03396	0.18428	0.12269	0.96603

’output_files_transfer_time’. The failure in accurately predicting ’compute_time’ could be attributed to the lack of variance in the training and evaluation data, which led the model to essentially guess at random. Additionally, all values of ’output_files_transfer_time’ in the training and evaluation set were very small for given input data (where all files have identical sizes), which might have contributed to the model’s poor performance in predicting this parameter.

Table 7.2.: GRU: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00283	0.05319	0.0369	0.99717
job_end	0.00315	0.05611	0.03939	0.99685
compute_time	2.19886	1.48285	1.10719	-1.19886
input_files_transfer_time	0.94084	0.96997	0.56842	0.05916
output_files_transfer_time	7.12789	2.66981	1.54578	-6.12789

7.2. LSTM

A total of 32 LSTM models were trained. The best LSTM model used one BiLSTM layer with a hidden size of 128, a window size of 50, no overlap, and a a batch size of 128. The windowing without overlap yielded the best results.

7.2.1. Interpolation

Figures 7.5 and 7.6, and Table 7.3 present the results achieved by the LSTM model in the interpolation task. The LSTM model exhibited performance similar to that of the GRU model in the interpolation task, successfully predicting all parameters with very high accuracy. It performed better than the GRU model in predicting the ’compute_time’ parameter, which is evident from Figures 7.5 and 7.1 for ’compute_time’ parameter. All distributions in KDE plots in Figures 7.6 are better than GRU.

Table 7.3.: LSTM: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00014	0.01203	0.00918	0.99986
job_end	0.00015	0.01205	0.0092	0.99985
compute_time	0.00595	0.07717	0.03803	0.99404
input_files_transfer_time	0.00062	0.02481	0.01601	0.99938
output_files_transfer_time	0.0093	0.09646	0.0584	0.99069

7.2.2. Extrapolation

Figures 7.7 and 7.8, and Table 7.4 present the results achieved by the LSTM model in the extrapolation task. The LSTM model performed similarly to the GRU model, effectively predicting 'job_start' and 'job_end' while failing to accurately forecast other parameters. The parameters 'input_files_transfer_time', 'compute_time' and 'output_files_transfer_time' are predicted with high error.

Table 7.4.: LSTM: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00452	0.06721	0.03989	0.99548
job_end	0.00488	0.06984	0.04229	0.99512
compute_time	2.07541	1.44063	1.0203	-1.07541
input_files_transfer_time	0.81828	0.90459	0.46304	0.18172
output_files_transfer_time	6.37867	2.5256	1.52316	-5.37867

7.3. Transformer

A total of 41 Transformer models were trained. The most effective Transformer model utilized a window size of 100 without overlap, a batch size of 128, a single encoder layer with a hidden size of 8, and one attention head.

7.3.1. Interpolation

Figures 7.9 and 7.10, and Table 7.5 present the results achieved by the Transformer model in the interpolation task. The Transformer model exhibited a poorer performance in matching the distribution compared to the GRU and LSTM models, particularly evident in its handling of the 'compute_time' parameter. Unlike its counterparts, the Transformer consistently predicted the average value of 'compute_time', which can be observed in

Figure 7.10 for the 'compute_time' parameter. The Transformer model also delivered poorer predictions for 'output_files_transfer_time' compared to the LSTM and GRU models. This is evident from the metrics for 'output_files_transfer_time' parameter presented in Table 7.5. The R-squared value for 'output_files_transfer_time' is 0.60185, while the LSTM showed the R-squared value of 0.99069 in Table 7.3.

Table 7.5.: Transformer: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.0012	0.03461	0.02745	0.9988
job_end	0.00124	0.03518	0.02783	0.99876
compute_time	0.91219	0.95508	0.78139	0.08765
input_files_transfer_time	0.01156	0.10754	0.039	0.98843
output_files_transfer_time	0.39808	0.63093	0.32893	0.60185

7.3.2. Extrapolation

Figures 7.11 and 7.12, and Table 7.6 present the results achieved by the Transformer model in the extrapolation task. The systematical overestimation for 'job_start' and 'job_end' values in for the first jobs and the underestimation for the last jobs in sequences can be observed in both accuracy and KDE plots. As in the interpolation task, the Transformer model predicted mostly the mean values for 'compute_time', and did not predicted the distribution, which can be observed in corresponding KDE plot in Figure 7.12. The predictions for 'input_files_transfer_time' and 'output_files_transfer_time' are not meaningful.

Table 7.6.: Transformer: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.0074	0.08602	0.0551	0.9926
job_end	0.00798	0.08934	0.05823	0.99202
compute_time	1.13928	1.06737	0.86759	-0.13928
input_files_transfer_time	0.92634	0.96247	0.47598	0.07366
output_files_transfer_time	1.83859	1.35595	0.97126	-0.83859

7.4. Models with more Layers

Figures 7.13 present the change in extrapolation accuracy with an increase in the number of model layers. In the plots, we observe that GRU and Transformer models with more layers predict different distributions compared to those with fewer layers. In contrast, LSTM models with varying numbers of layers produced similar predictions. Each model type with 2, 4, and 8 layers was trained and evaluated 10 times on the same dataset, and this behavior was consistent across different configurations. The interpolation accuracy of the models remained very high, showing no differences from the single-layer models discussed in previous sections.

7.5. Discussion

In the interpolation task, both the GRU and LSTM models demonstrated good accuracy in predictions of 'job_start', 'job_end', and 'compute_time'. For the extrapolation task, these models effectively predicted 'job_start' and 'job_end', with the GRU and LSTM models showing a slightly superior ability to match the actual distribution more accurately.

When a model is trained on a set of sequences with varying lengths, scaling features independently within each sequence leads to better predictions than scaling all sequences together. If the dataset contains sequences of varying lengths, such as one sequence with 10 jobs and another with 10.000 jobs, scaling these sequences together will result in disproportionately small values for the shorter sequence if all jobs share similar characteristics. This discrepancy arises because the scaling process normalizes the entire dataset based on the collective range and variance, which can overshadow the variability within shorter sequences.

In all three cases involving the GRU, LSTM, and Transformer models, a single-layer configuration yielded good results. Increasing the number of layers did not lead to significant improvement in extrapolation results. This suggests that, for this specific task, a simpler model architecture might be more effective than a more complex, multi-layered approach.

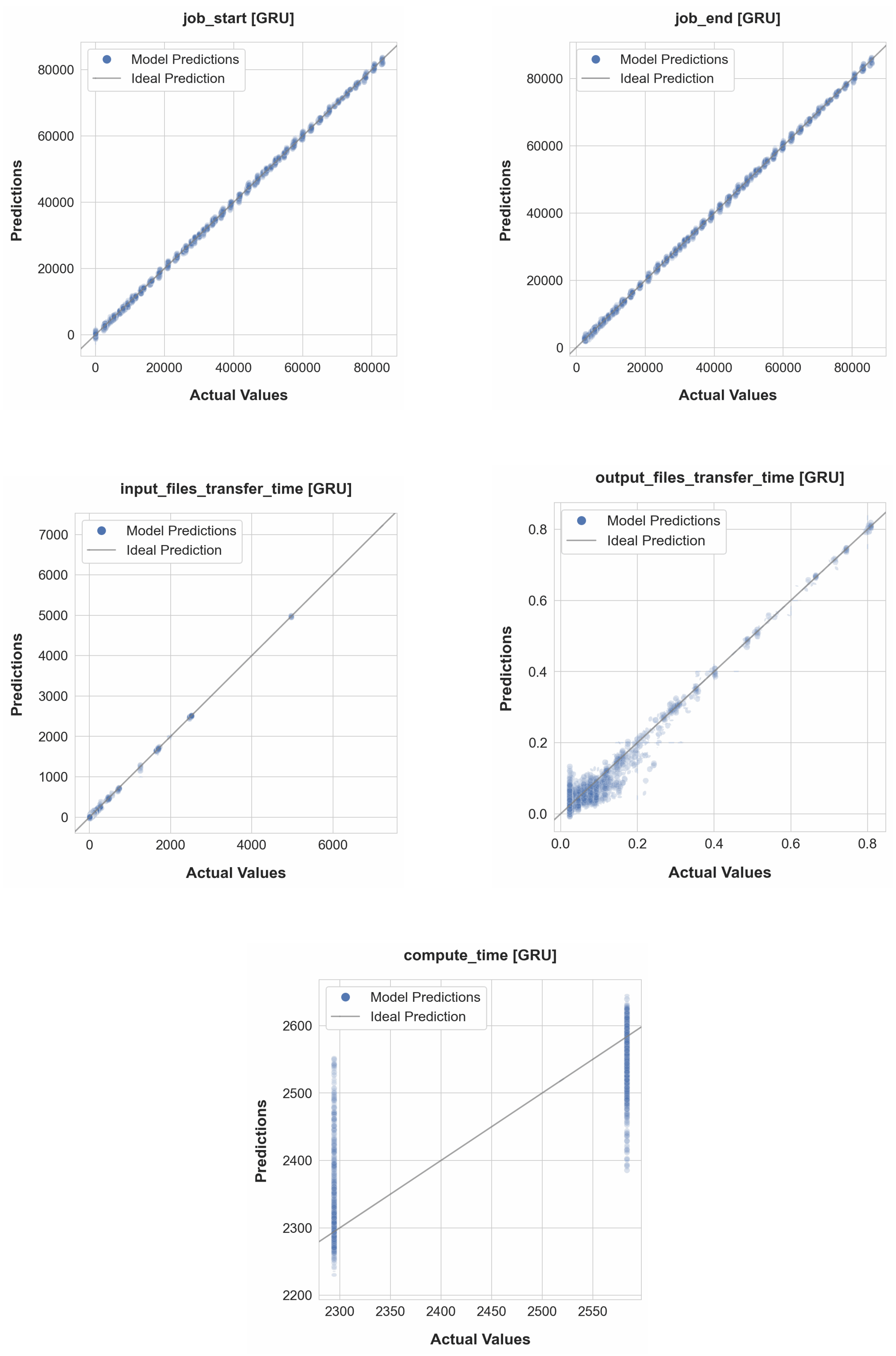


Figure 7.1.: GRU Interpolation Accuracy

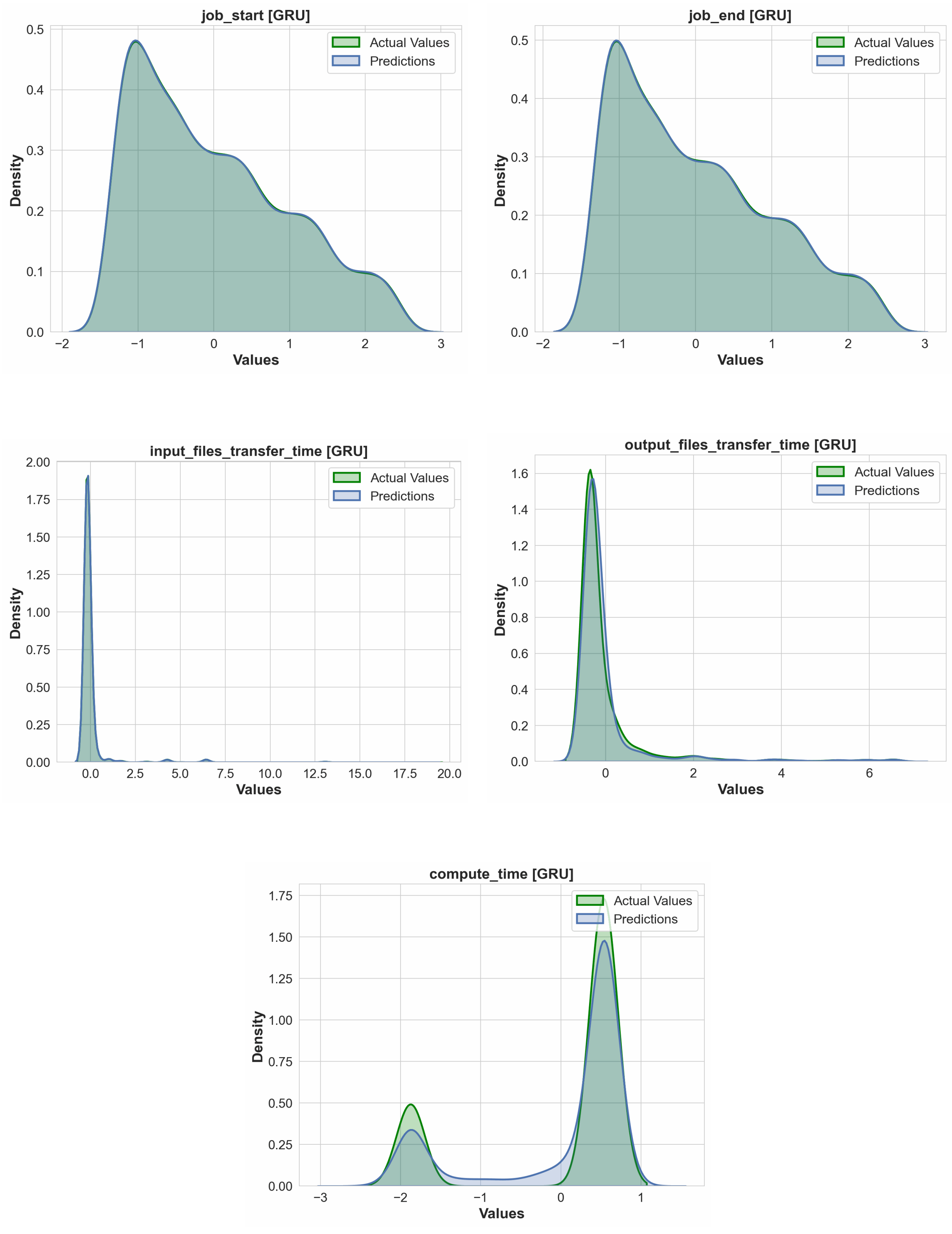


Figure 7.2.: GRU Interpolation KDE

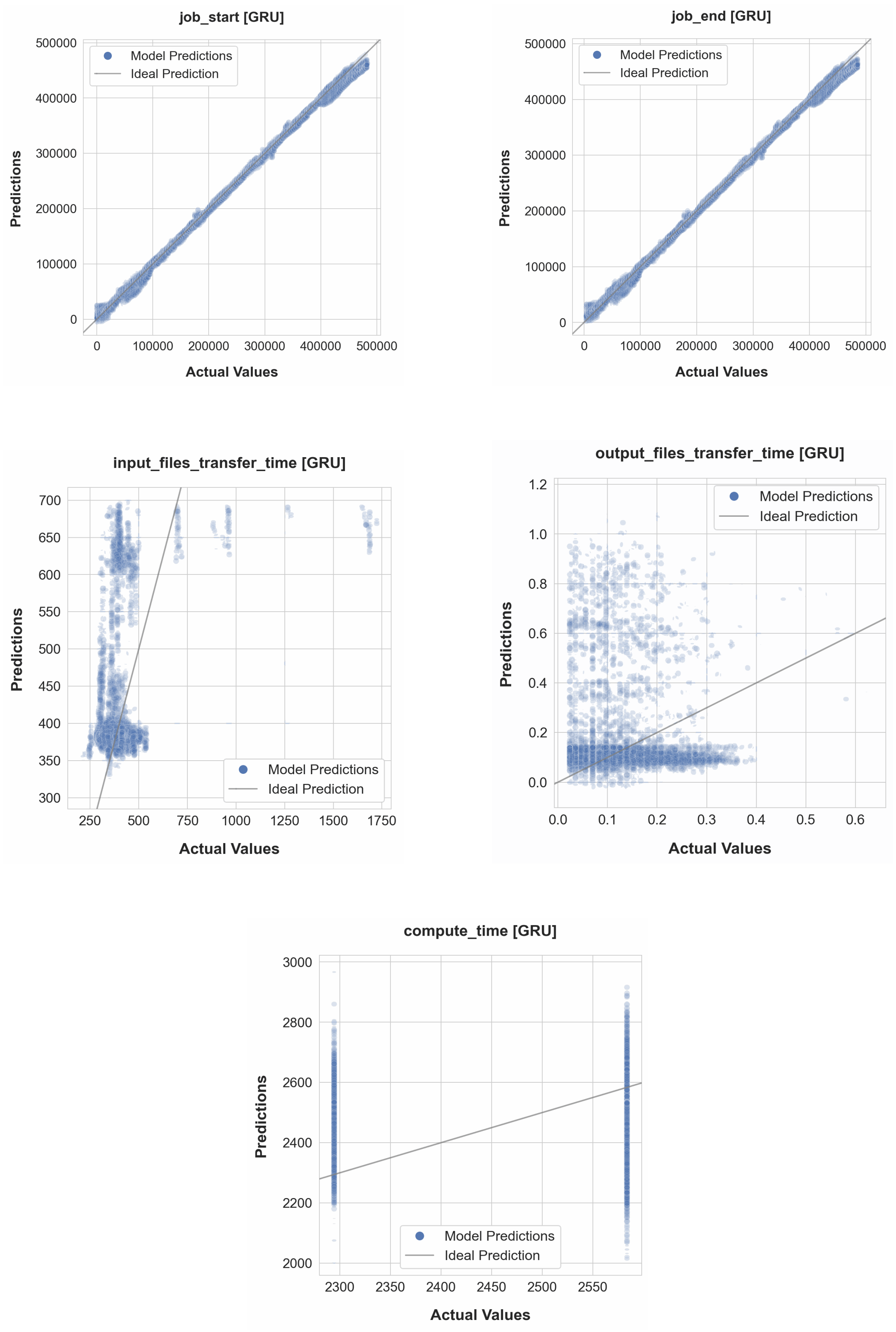


Figure 7.3.: GRU Extrapolation Accuracy

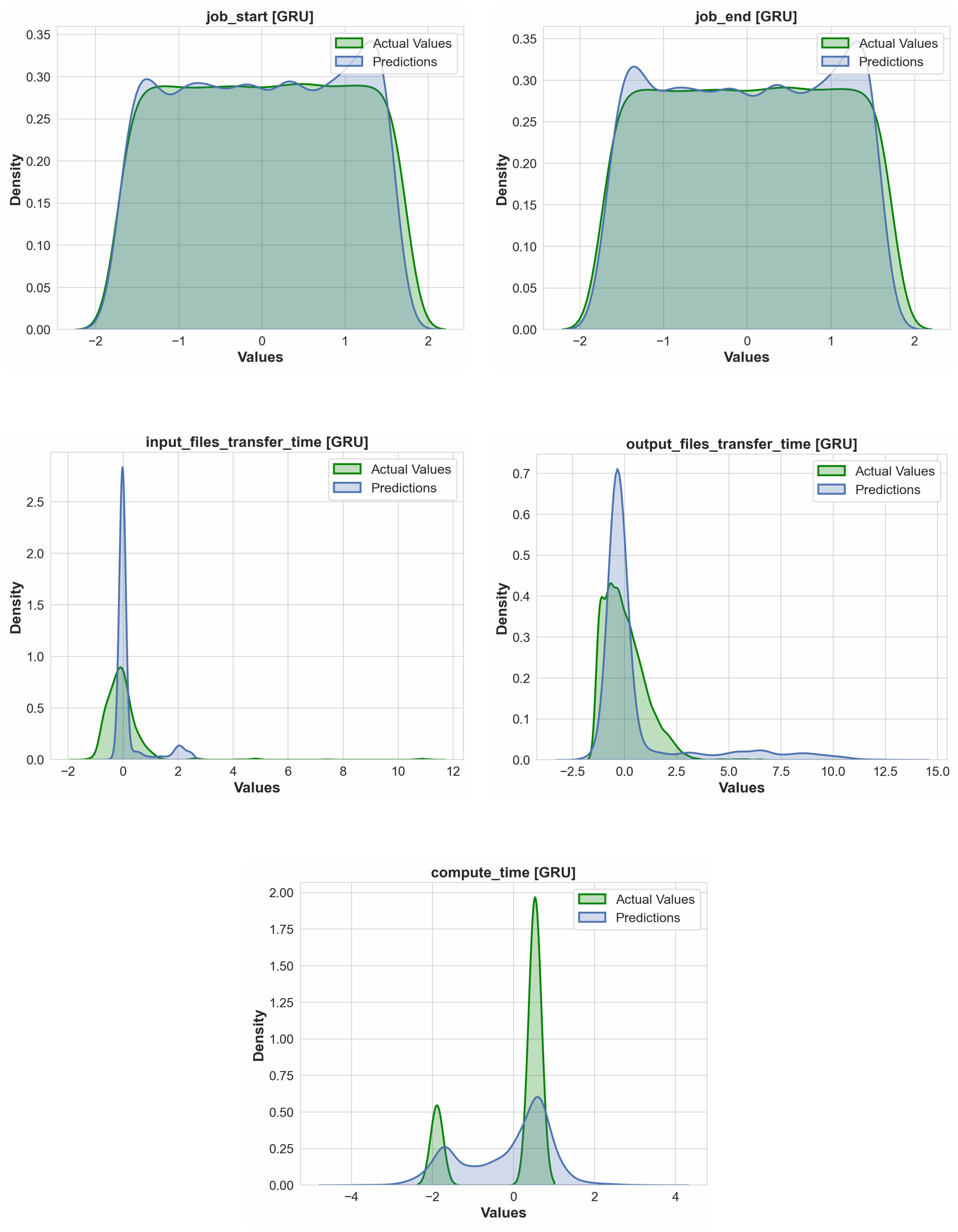


Figure 7.4.: GRU Extrapolation KDE

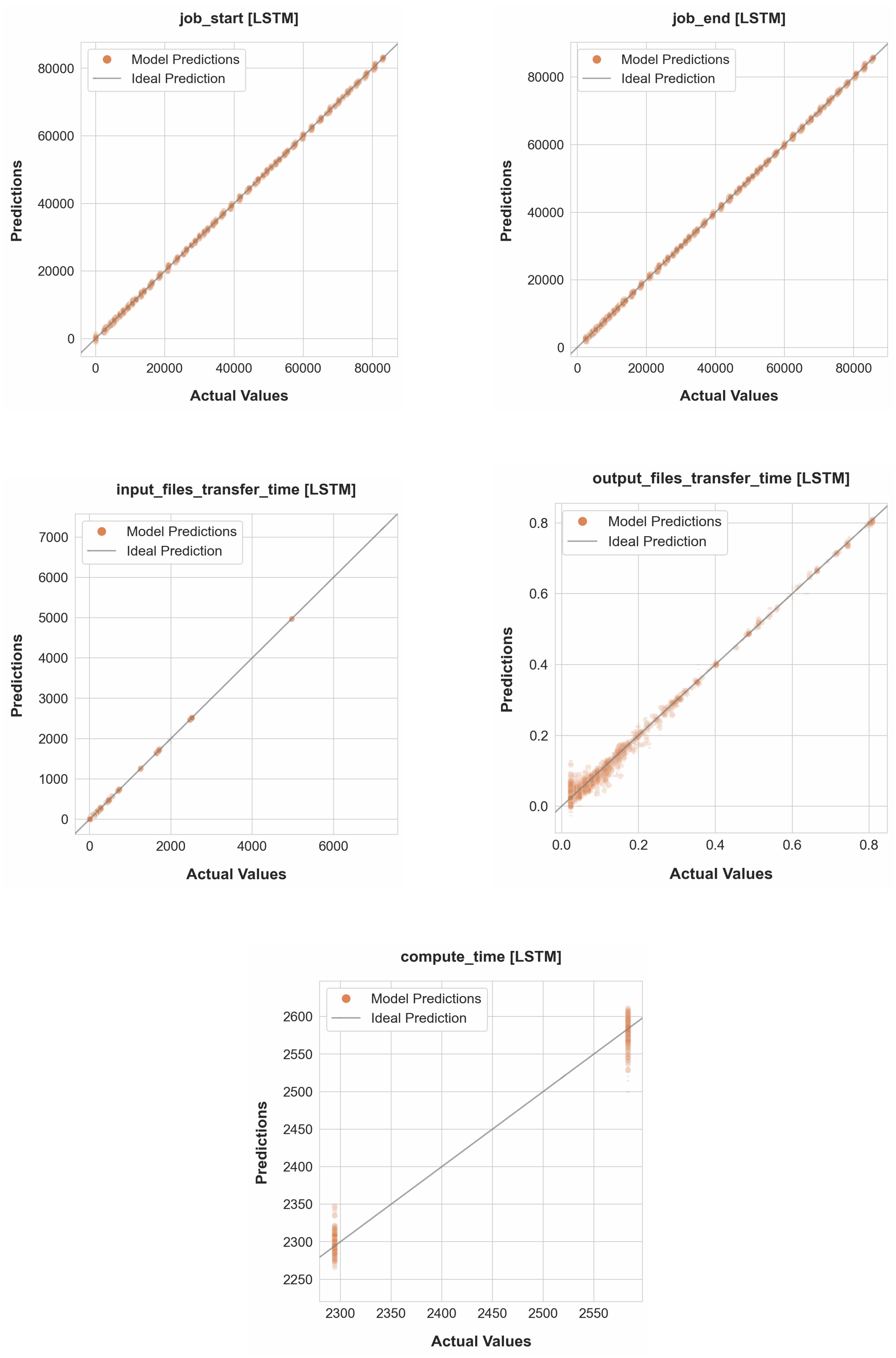


Figure 7.5.: LSTM Interpolation Accuracy

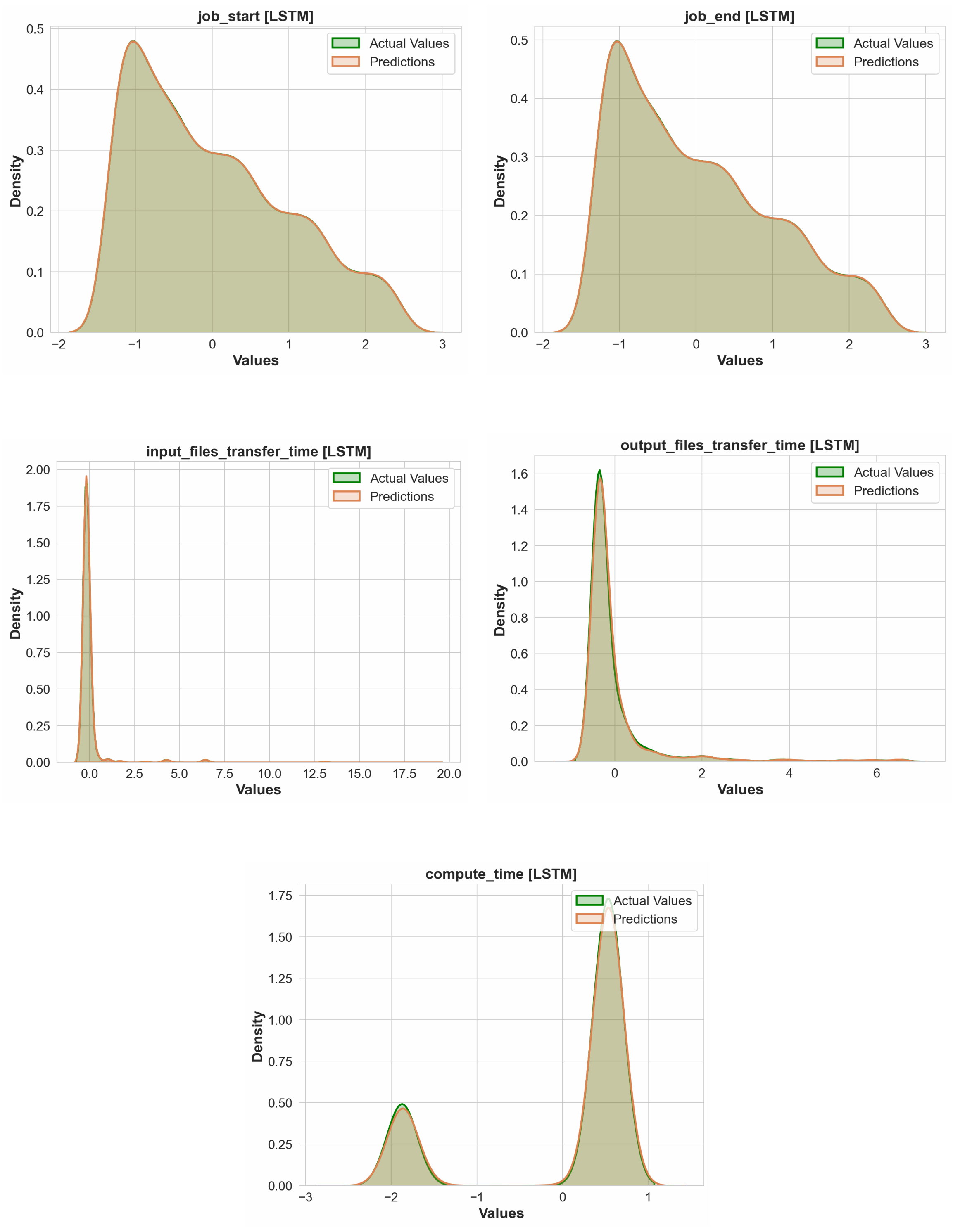


Figure 7.6.: LSTM Interpolation KDE

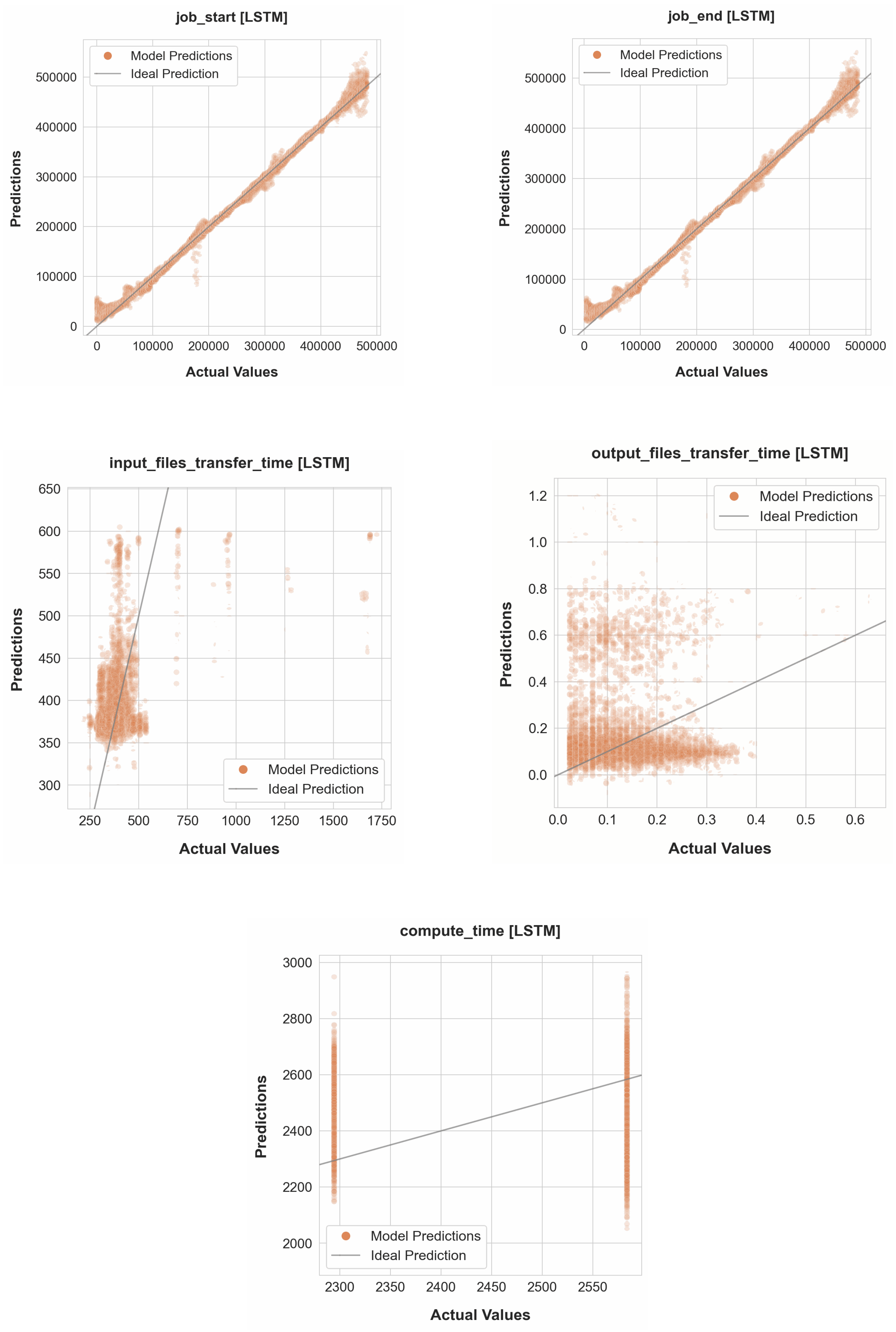


Figure 7.7.: LSTM Extrapolation Accuracy

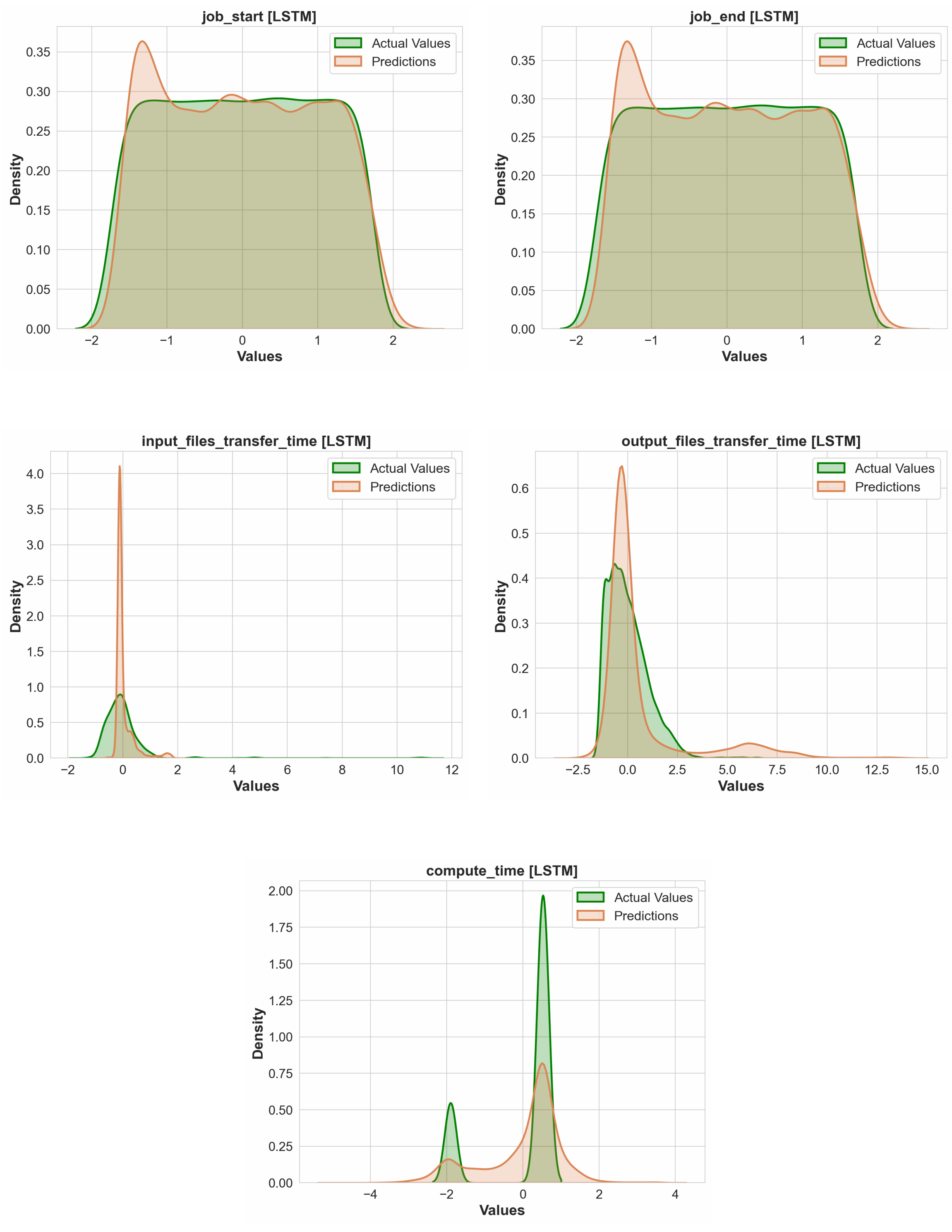


Figure 7.8.: LSTM Extrapolation KDE

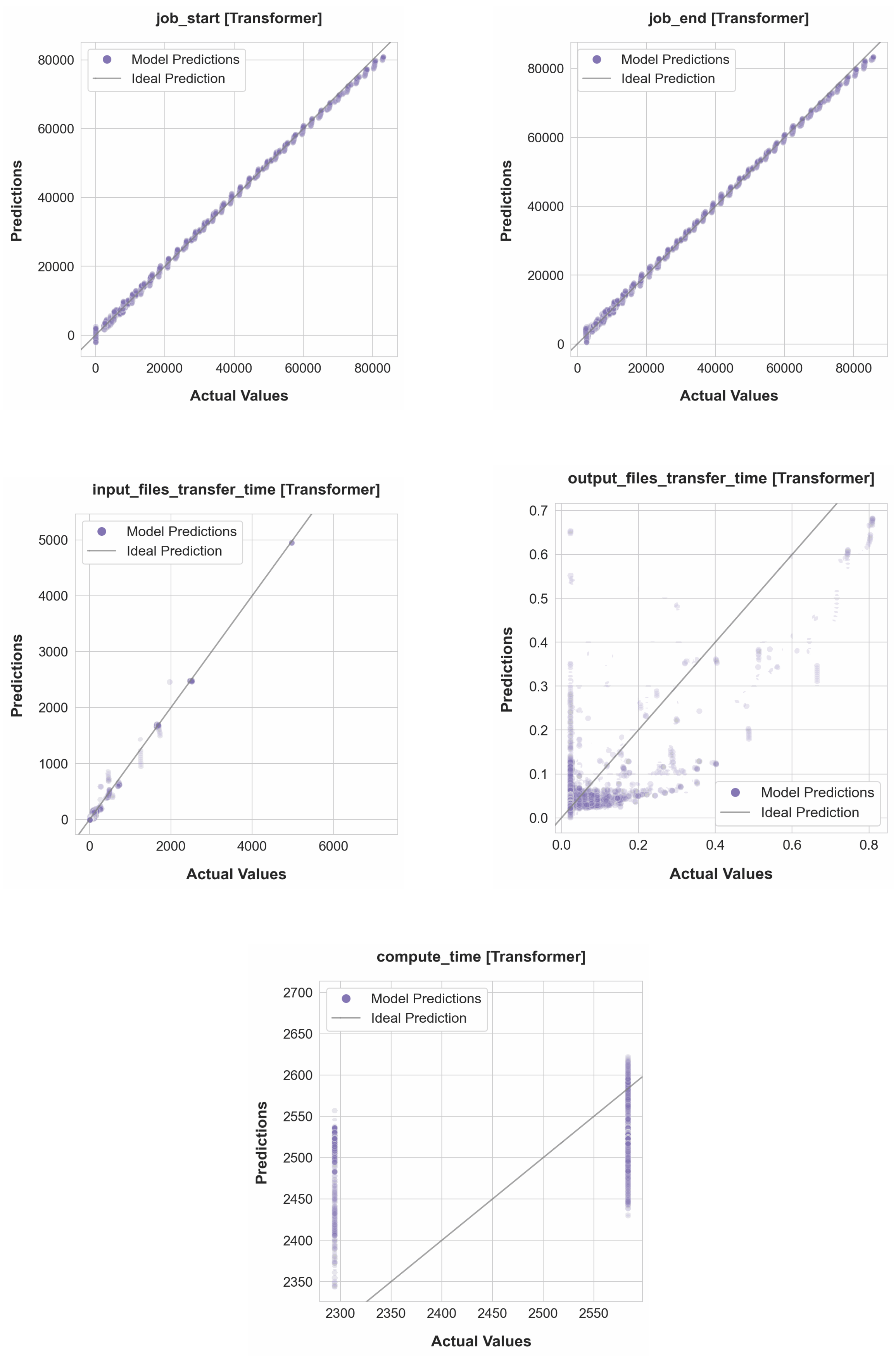


Figure 7.9.: Transformer Interpolation Accuracy

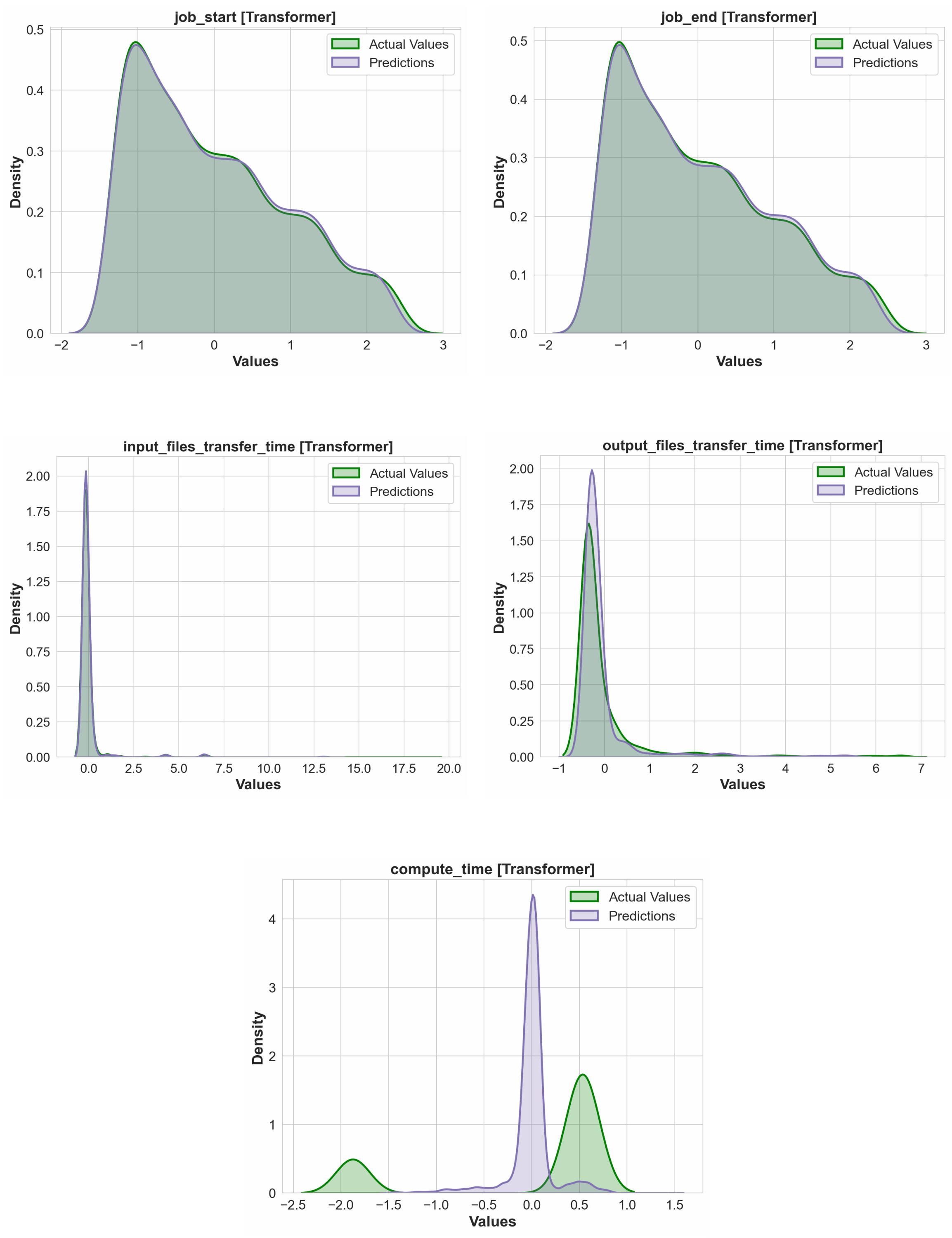


Figure 7.10.: Transformer Interpolation KDE

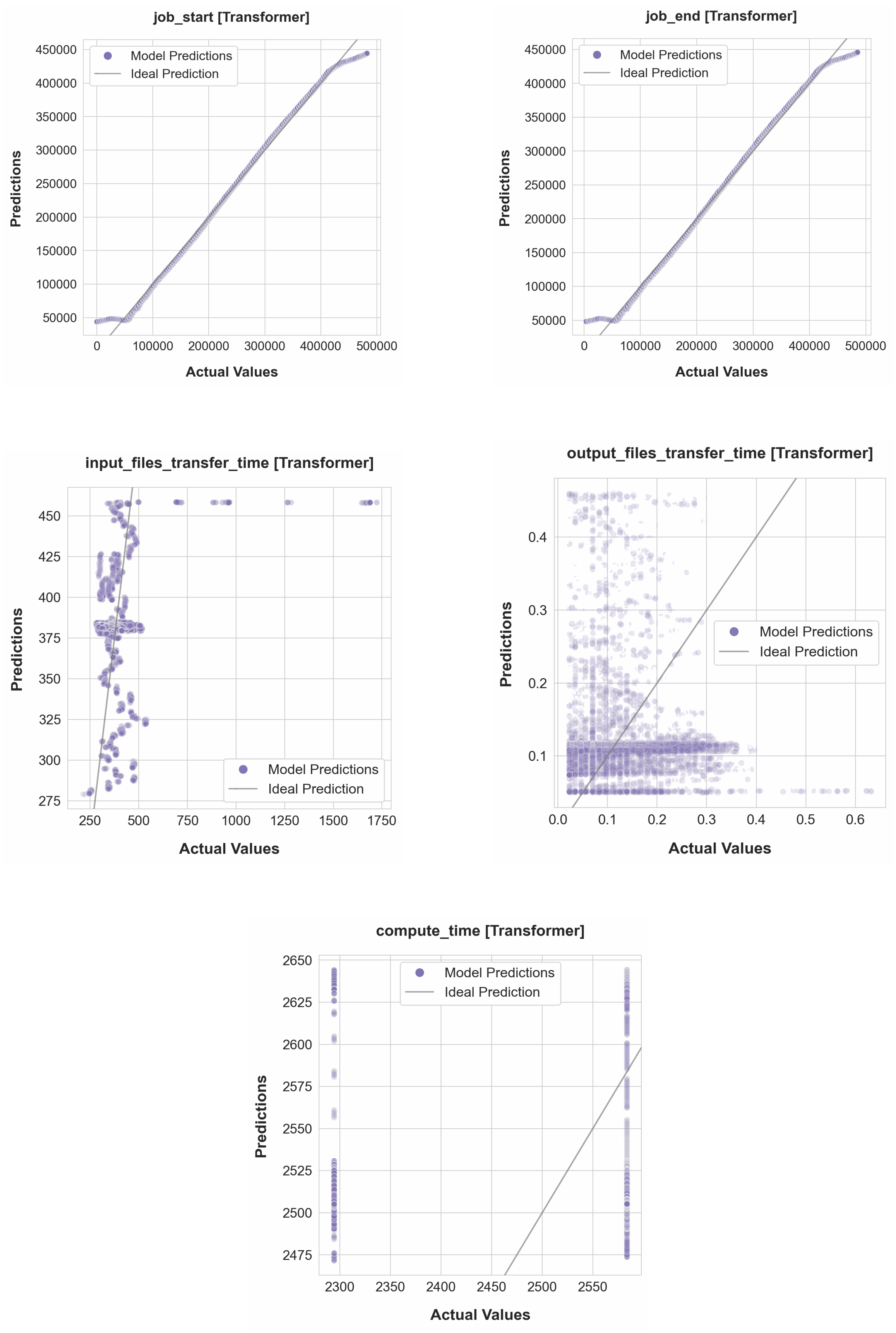


Figure 7.11.: Transformer Extrapolation Accuracy

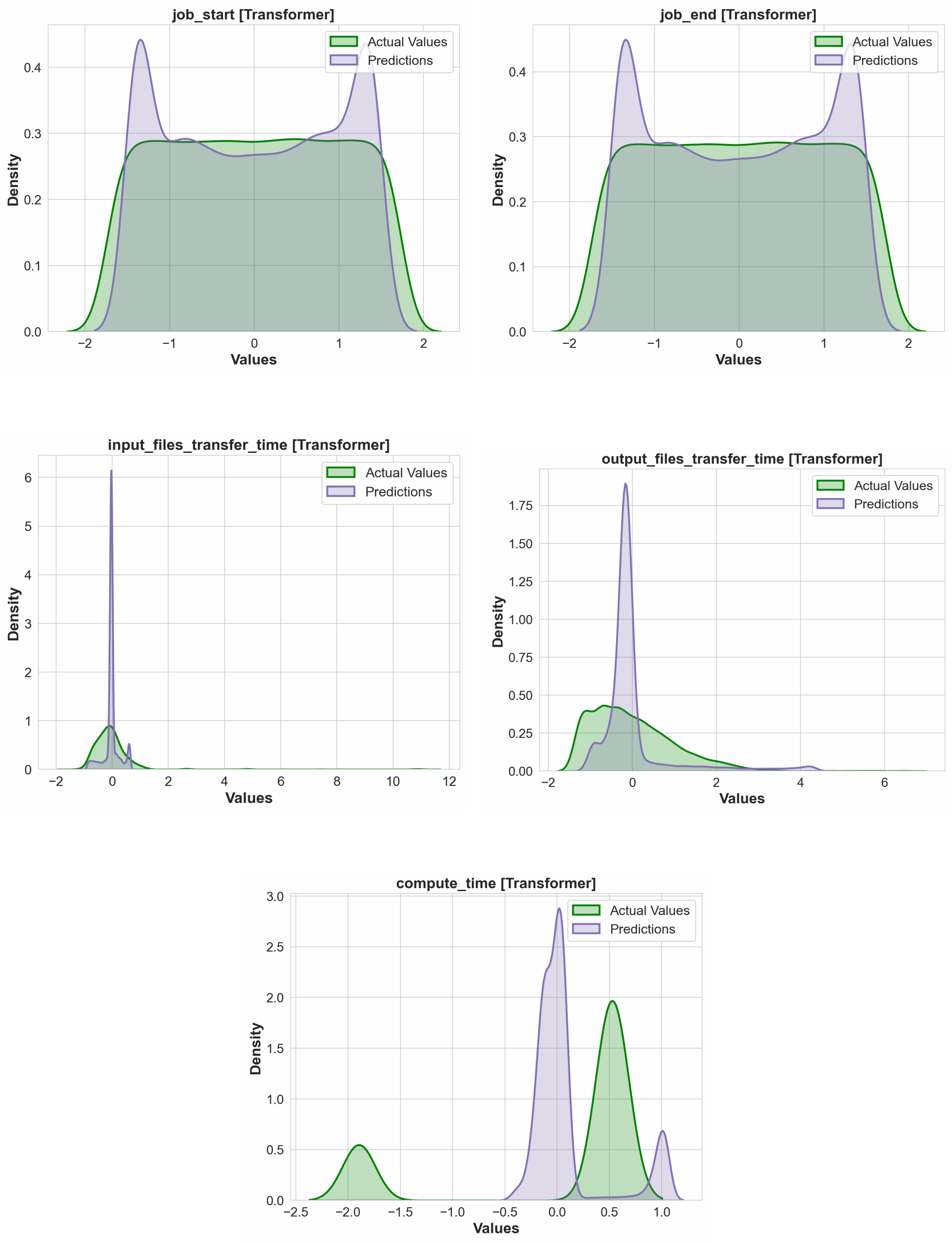
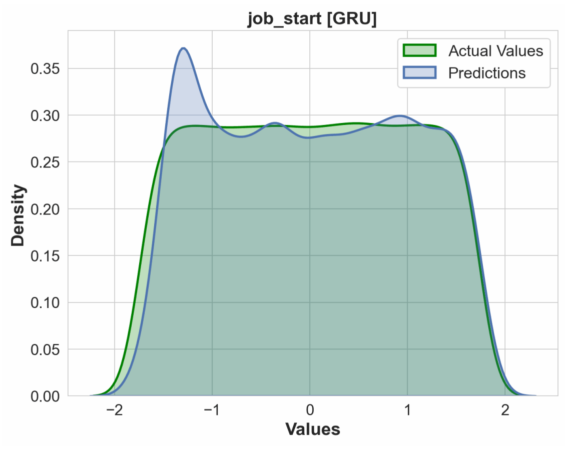
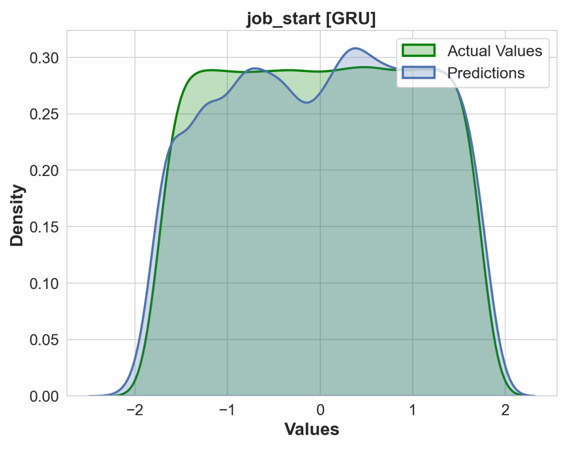


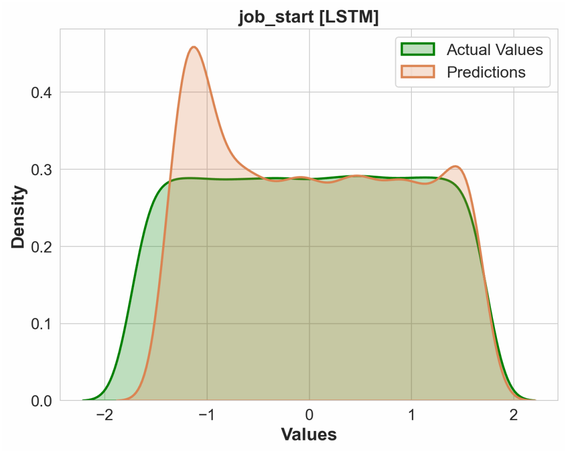
Figure 7.12.: Transformer Extrapolation KDE



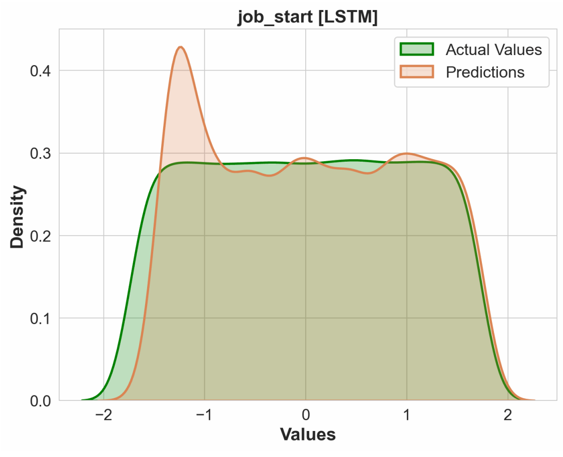
(a) GRU with 2 layers



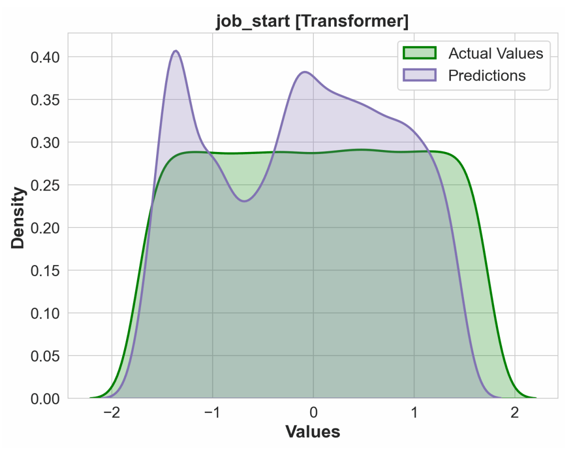
(b) GRU with 4 layers



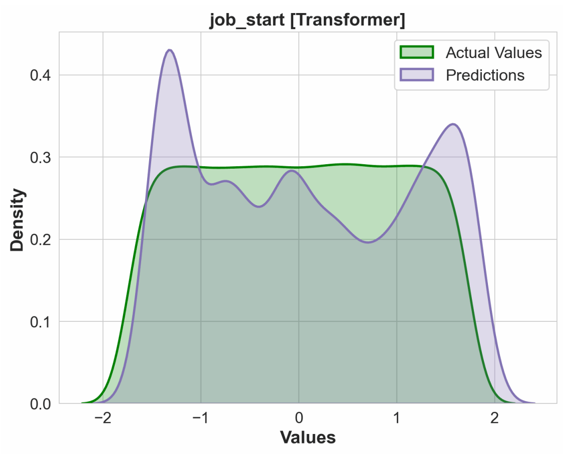
(c) LSTM with 2 layers



(d) LSTM with 4 layers



(e) Transformer with 2 layers



(f) Transformer with 4 layers

Figure 7.13.: Predictions of the models with multiple layers

8. Second Scenario: Heterogeneous Jobs

In contrast to the first scenario in our research, where the models were trained using jobs from a single uniform distribution, the second scenario introduces a more complex scenario. In this scenario, the models are trained to predict job execution parameters using jobs from five distinct job classes. Each class encompasses a unique statistical distribution of parameters, representing the real workloads of the WLCG. This approach allows the models to learn from a diverse range of job characteristics, mirroring the variability and complexity encountered in actual WLCG operations. The data preparation process for this scenario is described in Section 5.3.

8.1. GRU

A total of 38 GRU models were trained for this scenario. The best GRU model used one BiGRU layer with a hidden size of 16, which is noticeably less than in the first scenario. This model used a window size of 50, no overlap, and a batch size of 128.

8.1.1. Interpolation

Figures 8.1 and 8.2, and Table 8.1 present the results achieved by the GRU model in the interpolation task. GRU model was able to predict all parameters and match their distributions with very high accuracy, with R-squared near 0.99 for all parameters.

On the predictions plot for 'compute_time' in Figure 8.1 we can see the clear splitting of the data in two clusters. It appears that the model learned to make correct predictions for two nodes in the Sgbatch system, and slightly overestimates the 'compute_time' of computationally intensive jobs that are processed on the third, slower node.

8.1.2. Extrapolation

Figures 8.3 and 8.4, and Table 8.2 present the results achieved by the GRU model in the extrapolation task. The GRU model with a single layer demonstrated an ability to make accurate predictions in the extrapolation task. However, a slight overestimation of 'job_start' and 'job_end' was observed for sequence parts that exceeded previously seen sequence lengths. This tendency for overestimation is also evident in the KDE plots in Figure 8.4 for these parameters.

Table 8.1.: GRU: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00887	0.09417	0.06522	0.99113
job_end	0.00862	0.09283	0.06439	0.99138
compute_time	0.00414	0.06435	0.03624	0.99586
input_files_transfer_time	0.01776	0.13326	0.03716	0.98224
output_files_transfer_time	0.00755	0.08687	0.02385	0.99245

As observed in the interpolation task, some predictions for 'compute_time' were overestimated. However, this effect does not alter the overall distribution. This leads to the conclusion that only a very small proportion of the predictions were subject to overestimation.

The distribution for 'input_files_transfer_time' was not matched perfectly - an overestimation for smaller values of this parameter is evident in the KDE plot in Figure 8.4. The distribution of 'output_files_transfer_time' was matched perfectly.

Table 8.2.: GRU: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.07758	0.27852	0.20161	0.92242
job_end	0.08212	0.28657	0.20663	0.91788
compute_time	0.00546	0.07387	0.04571	0.99454
input_files_transfer_time	0.07009	0.26475	0.16799	0.92991
output_files_transfer_time	0.00648	0.08049	0.04661	0.99352

8.2. LSTM

A total of 27 LSTM models were trained for this scenario. The best LSTM model used one BiLSTM layer with a hidden size of 16, mirroring the behavior of GRU for the second time. This model used a window size of 50, no overlap, and a batch size of 128.

8.2.1. Interpolation

Figures 8.5 and 8.6, and Table 8.3 present the results achieved by the LSTM model in the interpolation task. The LSTM model demonstrated very good results in the interpolation task, with an accuracy within one percent of error coinciding with the GRU results. Similar to the GRU model, the LSTM model's predictions for 'compute_time' are also characterized by a division into two distinct clusters, which can be observed in Figure 8.5.

Table 8.3.: LSTM: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00866	0.09306	0.06343	0.99134
job_end	0.00842	0.09174	0.06266	0.99158
compute_time	0.00395	0.06283	0.03498	0.99605
input_files_transfer_time	0.01679	0.12956	0.02710	0.98321
output_files_transfer_time	0.00690	0.08309	0.01879	0.99310

8.2.2. Extrapolation

Figures 8.7 and 8.8, and Table 8.4 present the results achieved by the LSTM model in the extrapolation task. The LSTM model performed very similarly to GRU model.

Table 8.4.: LSTM: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.09527	0.30865	0.22686	0.90473
job_end	0.09955	0.31552	0.2309	0.90045
compute_time	0.00587	0.07664	0.04363	0.99413
input_files_transfer_time	0.12347	0.35139	0.21361	0.87653
output_files_transfer_time	0.00792	0.08897	0.04704	0.99208

8.3. Transformer

A total of 27 Transformer models were trained. The most effective Transformer model utilized a window size of 100 with an overlap of 30 elements. This is the first case when the overlap between windows leads to an improvement in prediction accuracy. However, the model with overlap 30 outperformed the second-best model which had no overlap only marginally, improving all metrics by 1,5%. The best Transformer model used a batch size of 128, a single encoder layer with a hidden size of 8, and four attention heads. An increase in the number of attention heads led to a better distribution of the predicted values, which was observed on KDE plots.

8.3.1. Interpolation

Figures 8.9 and 8.10, and Table 8.5 present the results achieved by the Transformer model in the interpolation task. The Transformer model demonstrated a very good capability in matching the actual values, mirroring the behavior of the GRU and LSTM models.

Table 8.5.: Transformer: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00910	0.09538	0.06542	0.99090
job_end	0.00881	0.09385	0.06444	0.99119
compute_time	0.00422	0.06493	0.03525	0.99578
input_files_transfer_time	0.03934	0.19836	0.02278	0.96066
output_files_transfer_time	0.00683	0.08267	0.01725	0.99317

8.3.2. Extrapolation

Figures 8.11 and 8.12, and Table 8.6 present the results achieved by the Transformer model in the extrapolation task. Same as in the previous task, Transformer model performed very similarly to GRU and LSTM models.

Table 8.6.: Transformer: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.0861	0.29342	0.21106	0.9139
job_end	0.08316	0.28838	0.20875	0.91683
compute_time	0.00742	0.08613	0.05379	0.99258
input_files_transfer_time	0.11493	0.33901	0.23491	0.88507
output_files_transfer_time	0.00685	0.08276	0.0471	0.99315

8.4. Discussion

All three models exhibited high accuracy in both interpolation and extrapolation tasks. Consistent with the first scenario, the best-performing models were the simple rather than complex ones with a lot of layers. Different types of model architectures demonstrate similar behavior, which could be caused by similar training strategies.

The optimal GRU and LSTM models have smaller hidden sizes than the models in the first scenario. In contrast, the Transformer model benefited from the use of window overlaps. The optimal GRU model had the same hyperparameters as the optimal LSTM in this scenario.

All three models accomplished the interpolation task with very similar results. The GRU model demonstrated the best results in the extrapolation task, according to accuracy metrics in Table 8.2.

The results of this scenario show, that the more complex parameter distributions could be captured with models with smaller hidden size. It is likely, that the greater hidden size used in the first scenario was necessitated by the limited variance present in the data, compared to the second scenario.

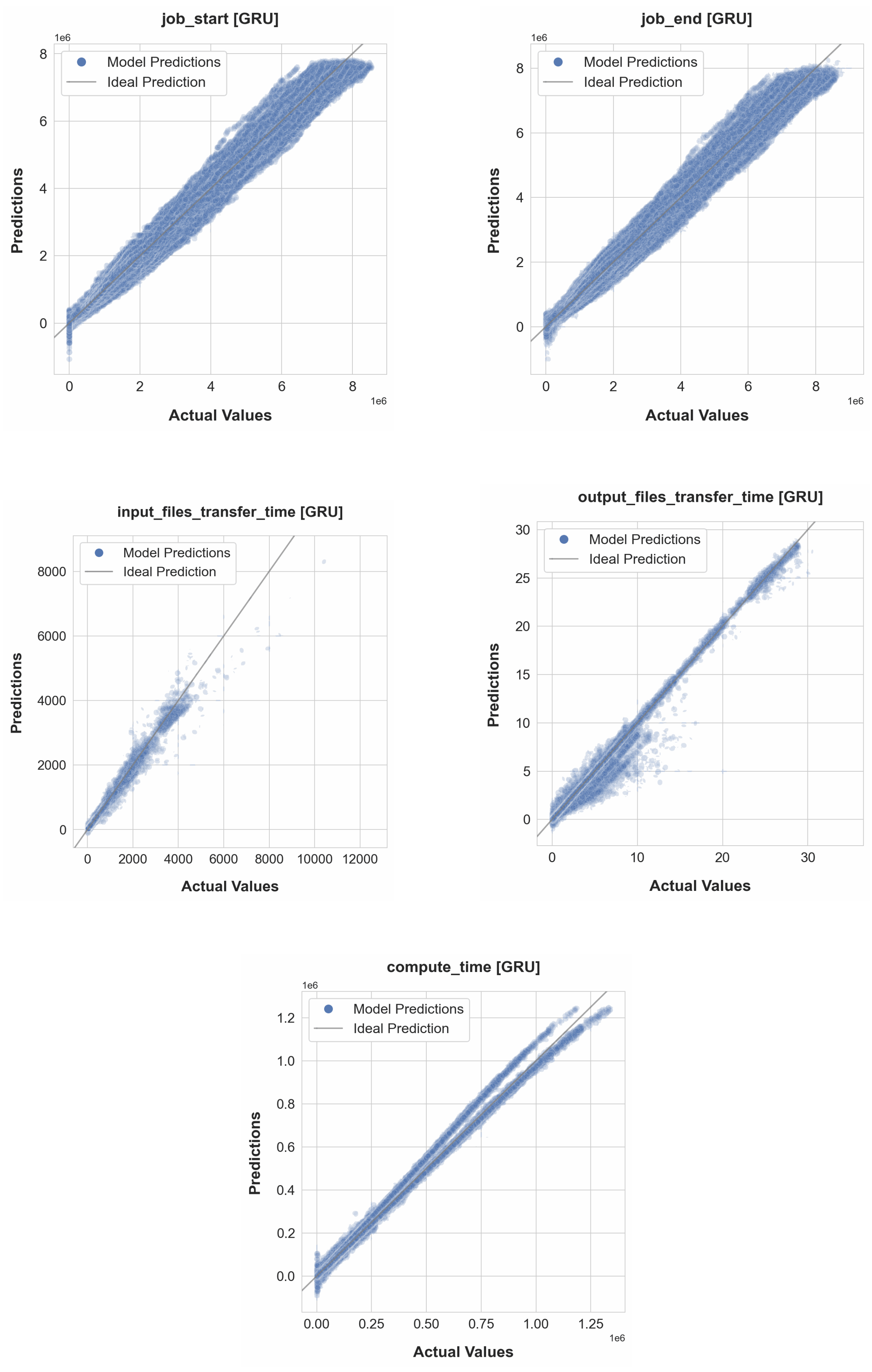


Figure 8.1.: GRU Interpolation Accuracy

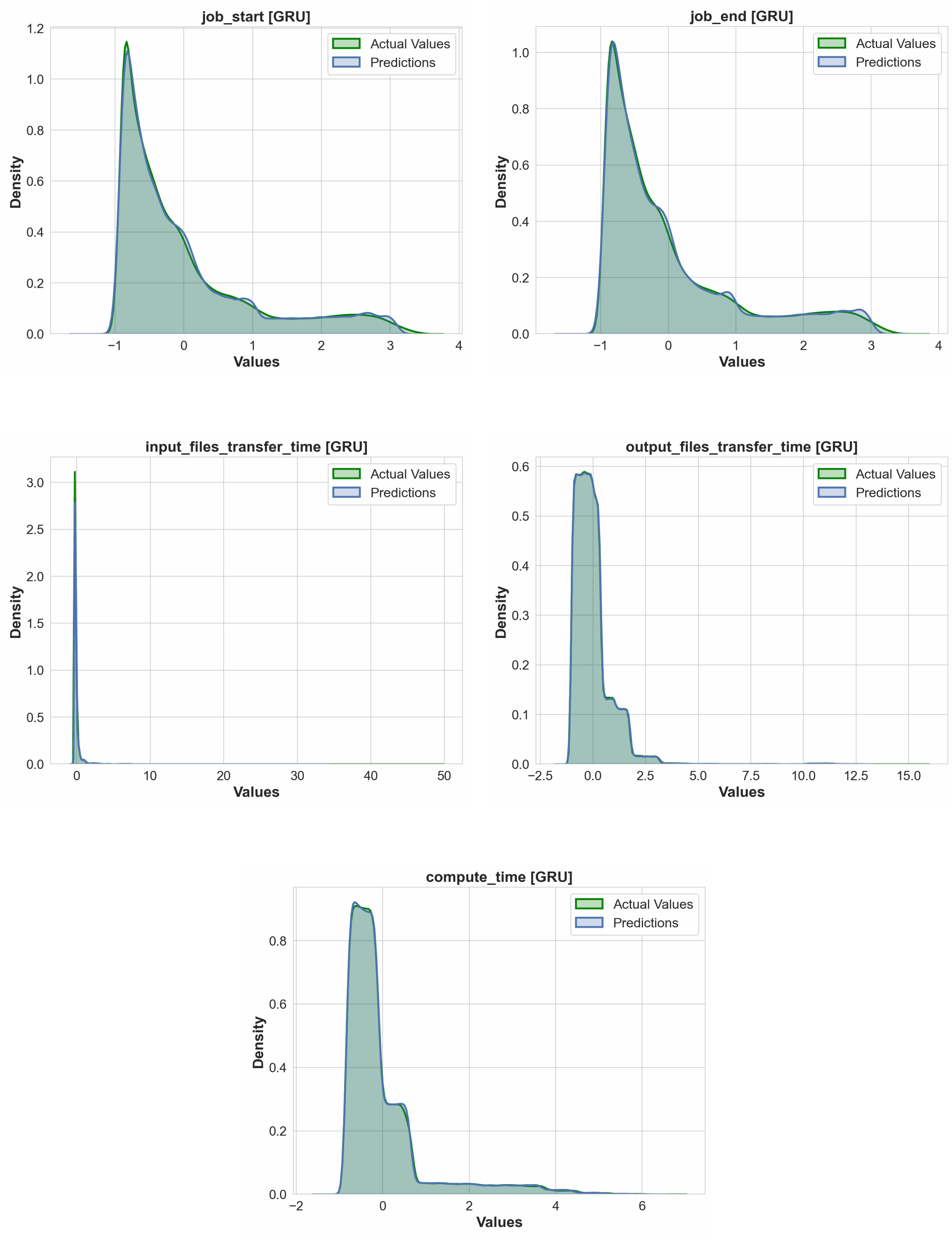


Figure 8.2.: GRU Interpolation KDE

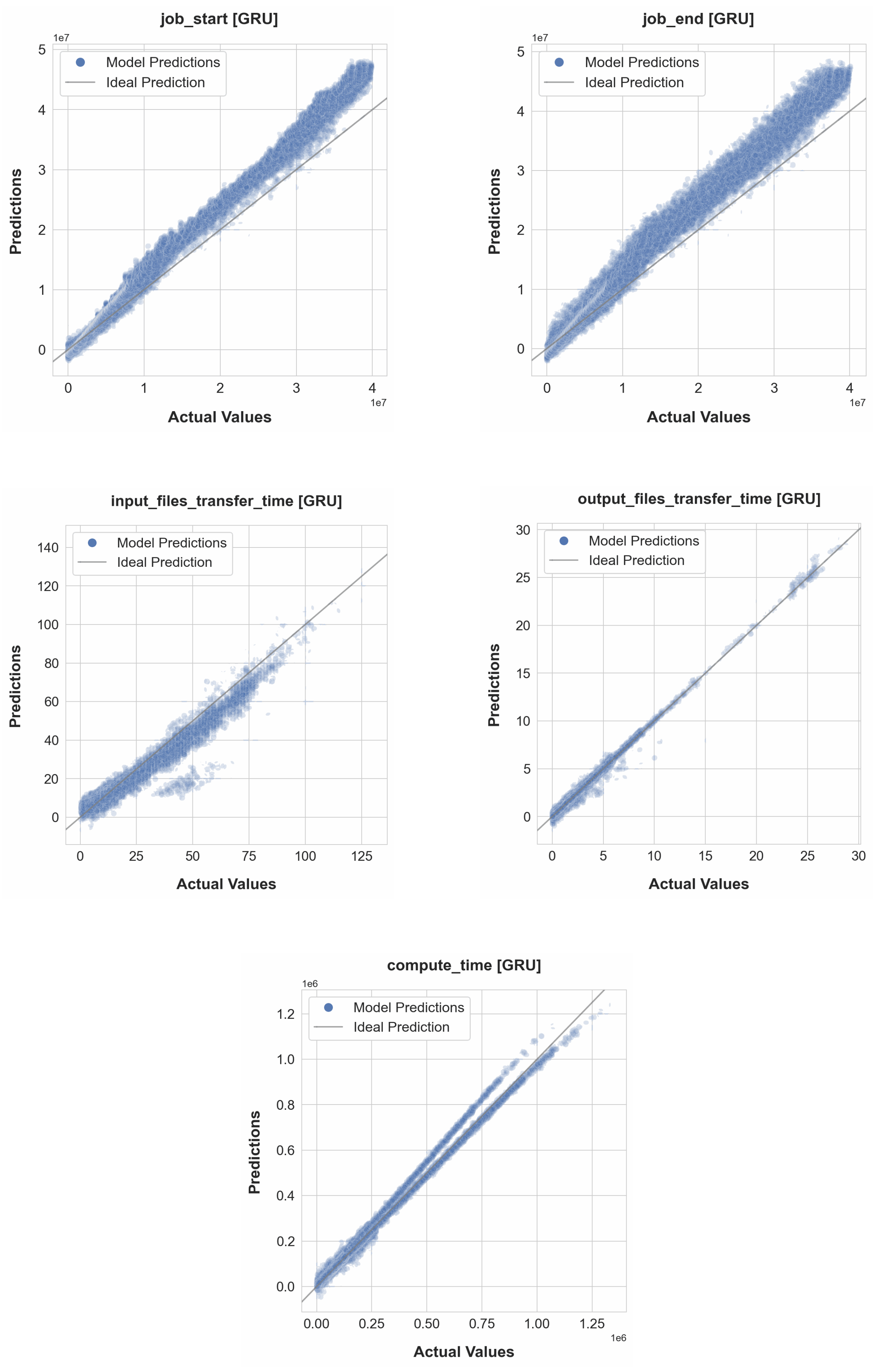


Figure 8.3.: GRU Extrapolation Accuracy

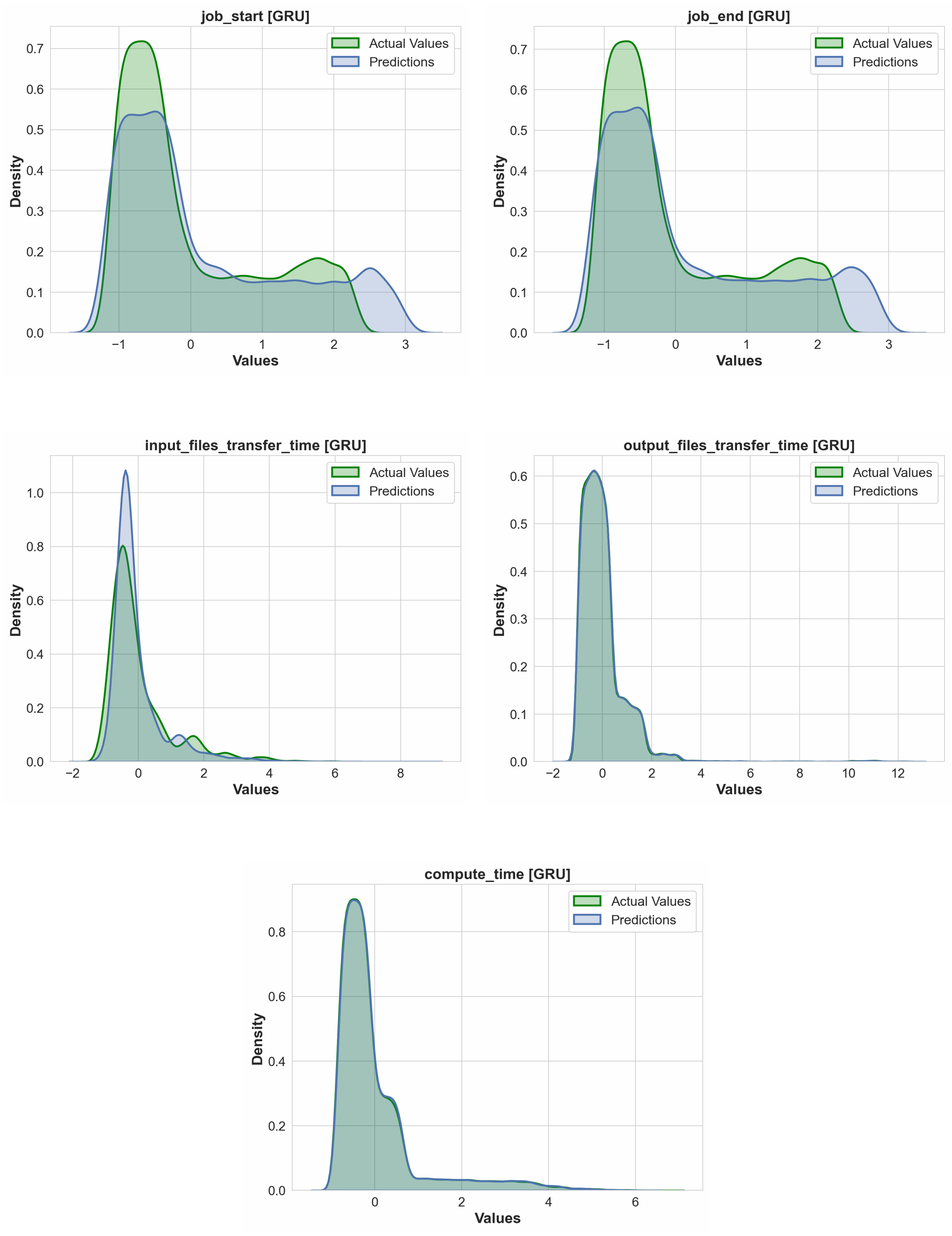


Figure 8.4.: GRU Extrapolation KDE

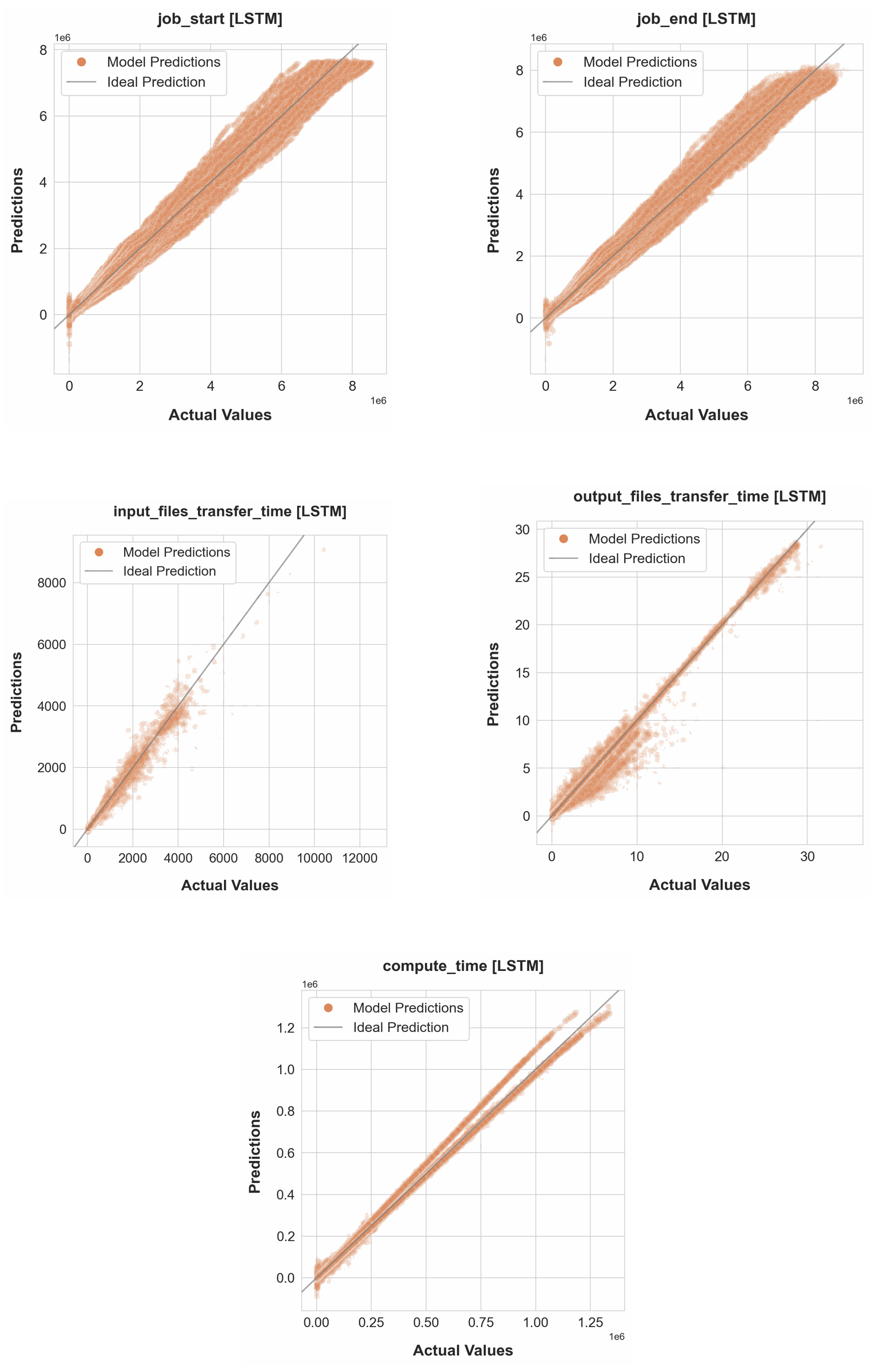


Figure 8.5.: LSTM Interpolation Accuracy

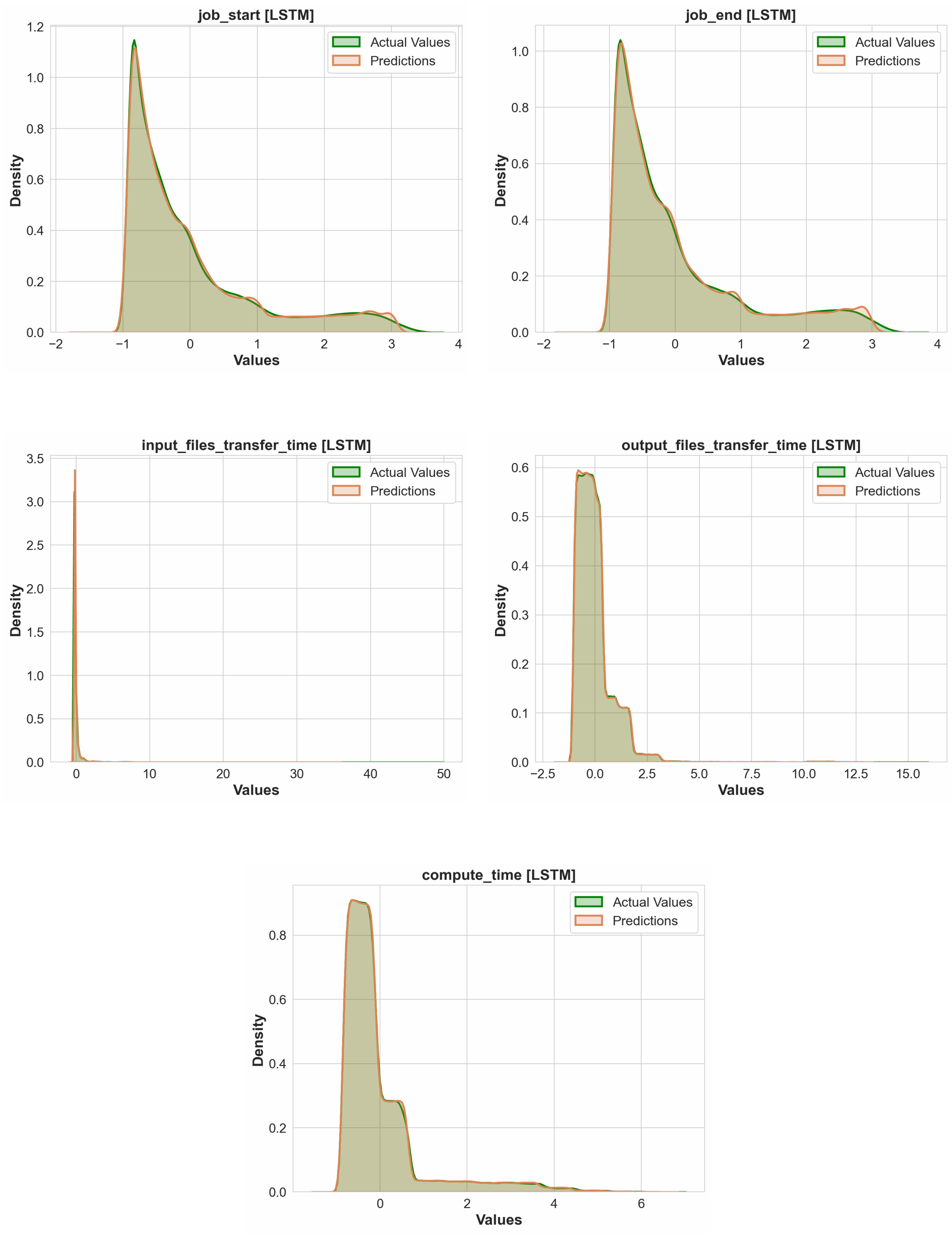


Figure 8.6.: LSTM Interpolation KDE

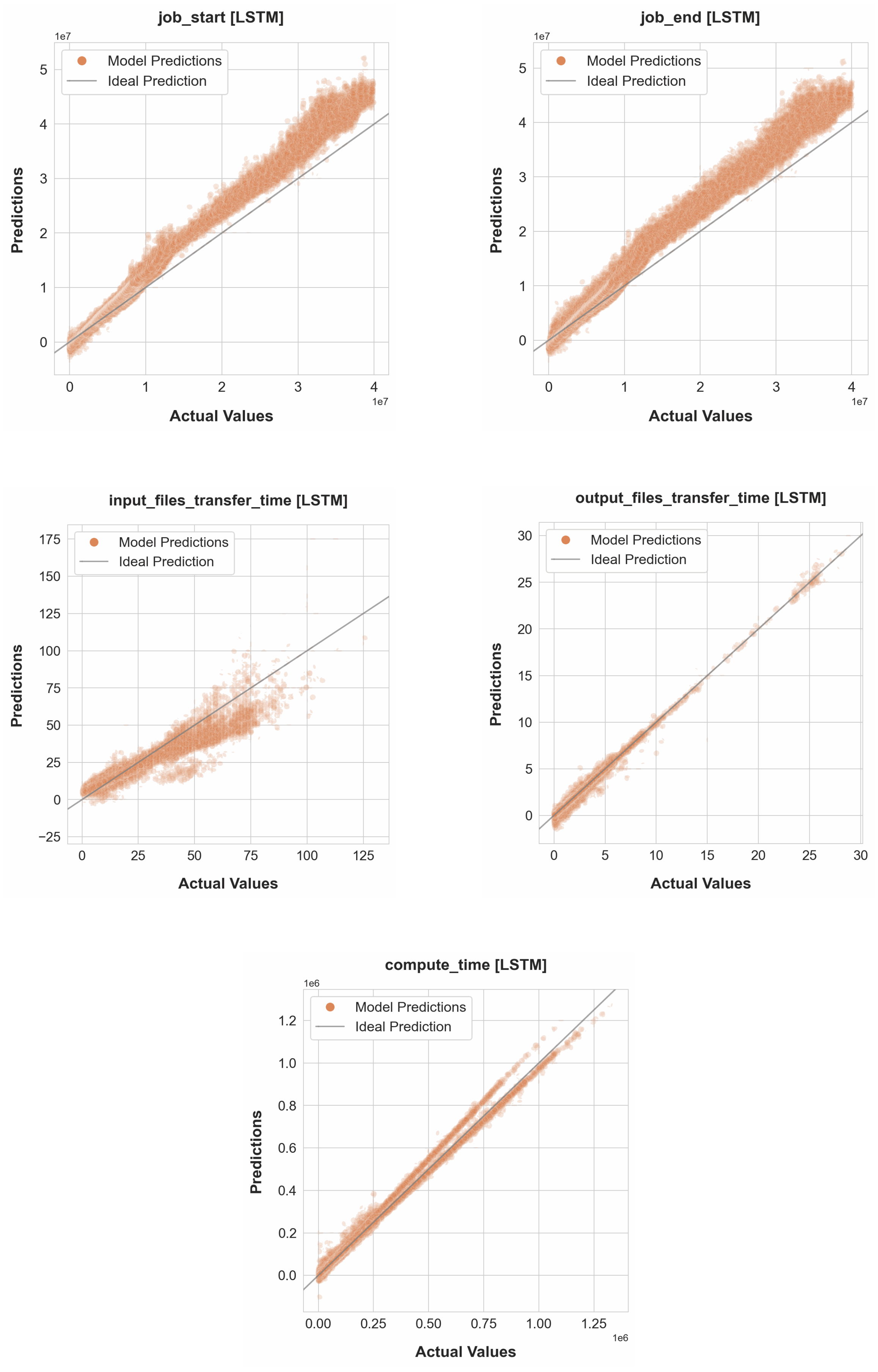


Figure 8.7.: LSTM Extrapolation Accuracy

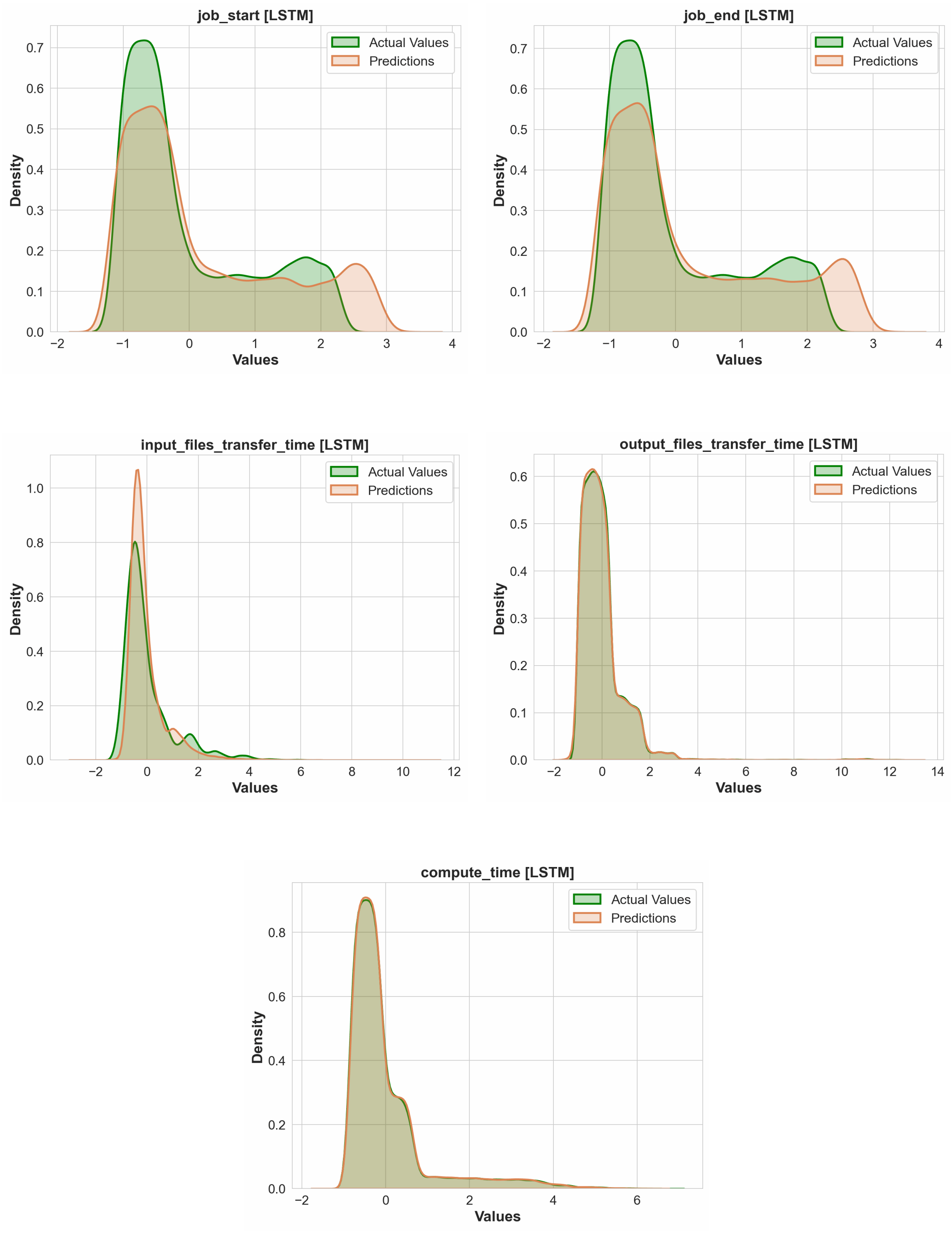


Figure 8.8.: LSTM Extrapolation KDE

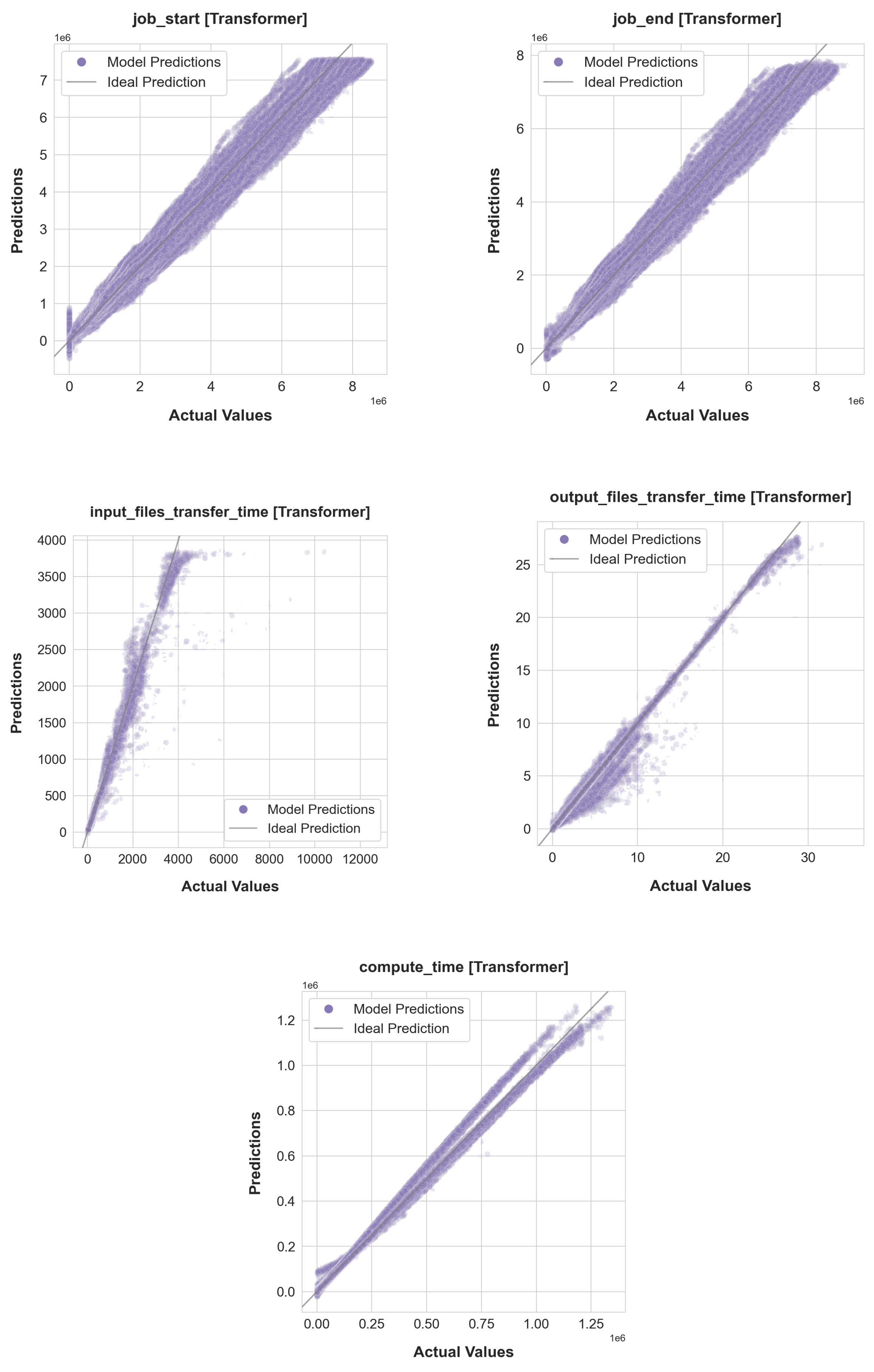


Figure 8.9.: Transformer Interpolation Accuracy

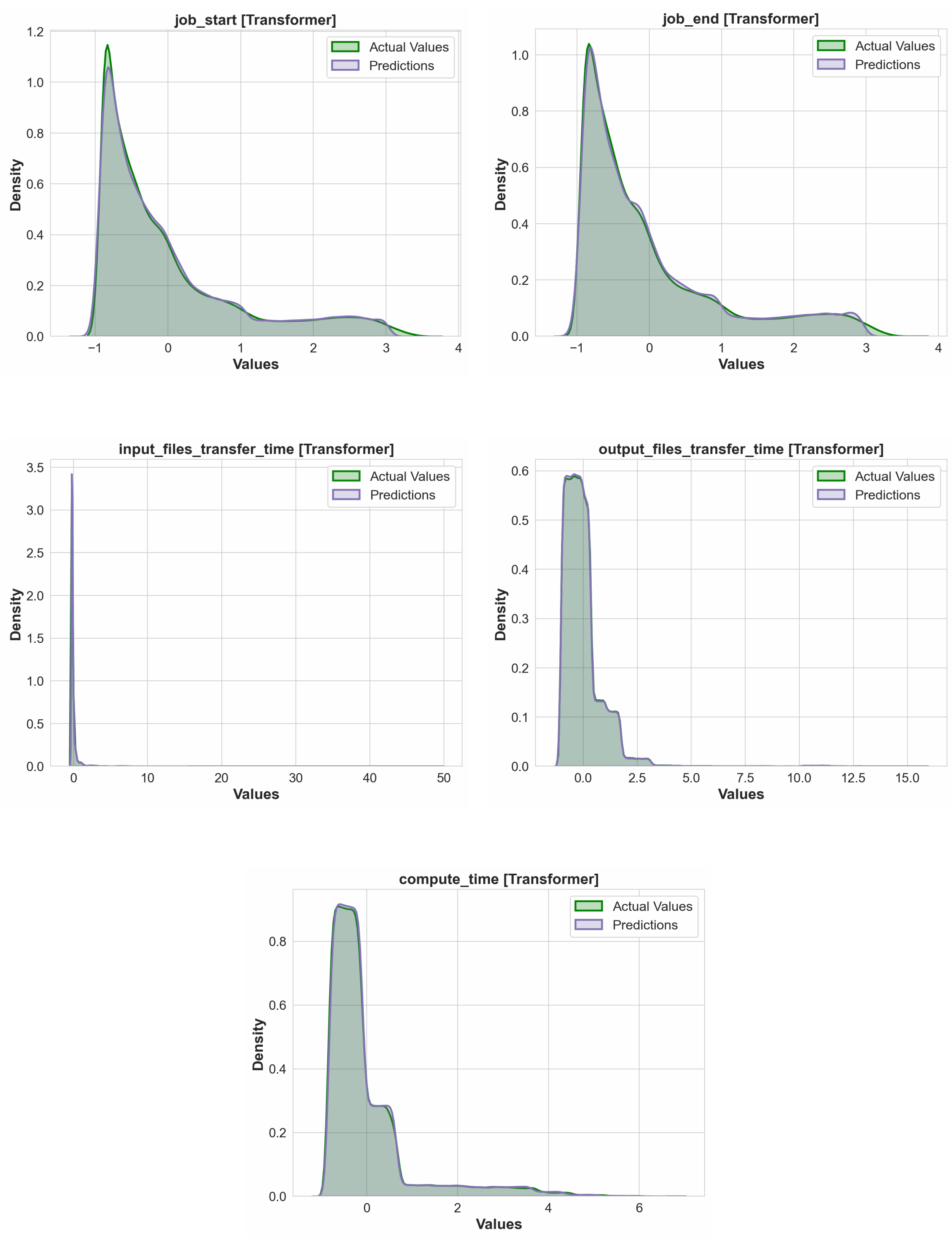


Figure 8.10.: Transformer Interpolation KDE

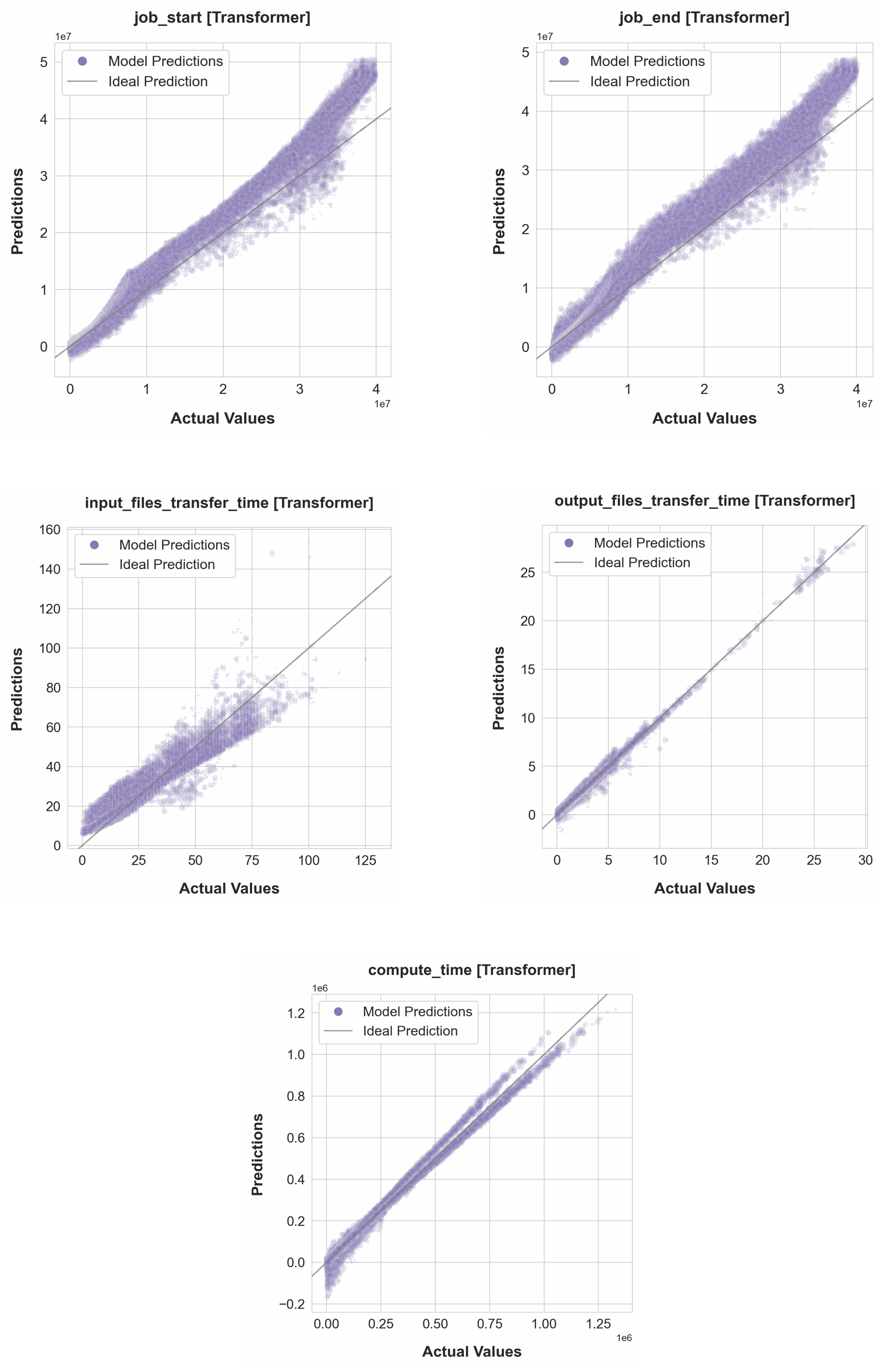


Figure 8.11.: Transformer Extrapolation Accuracy

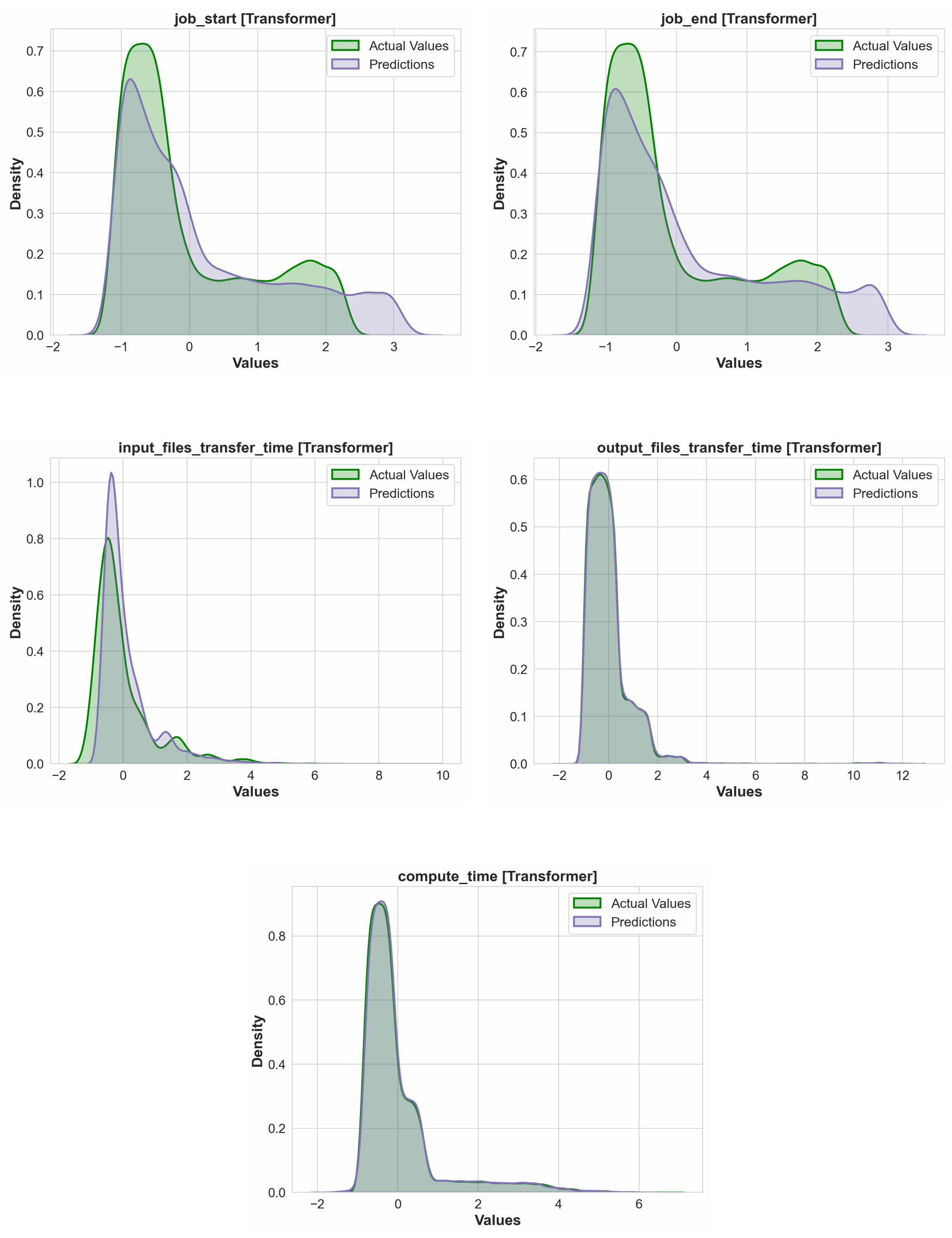


Figure 8.12.: Transformer Extrapolation KDE

9. Third Scenario: GridKA Platform

In this scenario, we generate jobs from five different job classes, similar to the second scenario. However, instead of utilizing the simpler Sgbatch system, we conduct our simulations on a more complex platform topology. Specifically, we focus on the interconnected Tier1 data center GridKa and the Tier2 computing center at DESY. The objective of this scenario is to examine the ability of our models to accurately predict simulation results for complex platforms. The data preparation process for this scenario is described in Section 5.4.

9.1. GRU

The best GRU model used one BiGRU layer with a hidden size of 128, a window size of 150, no overlap, and a batch size of 64.

9.1.1. Interpolation

Figures 9.1 and 9.2, and Table 9.1 present the results achieved by the GRU model in the interpolation task. GRU model predicted the 'job_start', 'job_end' and 'compute_time' parameters with a very high accuracy, which is evident from high R-squared and low errors in Table 9.1 and corresponding KDE plots. Metrics for 'input_files_transfer_time' and output_files_transfer_time' demonstrate higher error. KDE plots for these parameters in Figure 9.2 show, that the distributions are very narrow, even narrower than in the first scenario.

Table 9.1.: GRU: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.00562	0.075	0.05056	0.99438
job_end	0.01065	0.10318	0.06931	0.98935
compute_time	0.00615	0.07842	0.04198	0.99385
input_files_transfer_time	0.37285	0.61062	0.29772	0.62715
output_files_transfer_time	0.2736	0.52306	0.20744	0.7264

9.1.2. Extrapolation

Figures 9.3 and 9.4, and Table 9.2 present the results achieved by the GRU model in the extrapolation task. The predictions for 'job_start', 'job_end' and 'compute_time' have high accuracy according to metrics, but only the distribution for 'compute_time' is matched on average, which is evident from the plot in Figure 9.4. The accuracy plot for 'compute_time' in Figure 9.3 shows the forming of clusters with different deviation trends from the actual data. That could again be explained by the lack of information about the simulated hardware platform.

The systematic overestimation for 'job_start' and 'job_end' can be observed in Figures 9.3. The GRU model failed to predict the 'input_files_transfer_time' and 'output_files_transfer_time' parameters - the R-squared values for these parameters are negative.

Table 9.2.: GRU: Extrapolation accuracy metrics

Parameter	**MSE**	**RMSE**	**MAE**	**R^2**
job_start	0.02331	0.15269	0.11951	0.97669
job_end	0.13006	0.36063	0.25102	0.86994
compute_time	0.01259	0.1122	0.06183	0.98741
input_files_transfer_time	1.27119	1.12747	0.46941	-0.27119
output_files_transfer_time	1.1233	1.05986	0.42473	-0.1233

9.2. LSTM

The best LSTM model used one BiLSTM layer with a hidden size of 128, a window size of 150, no overlap, and a batch size of 64.

9.2.1. Interpolation

Figures 9.5 and 9.6, and Table 9.3 present the results achieved by the LSTM model in the interpolation task. The LSTM model demonstrated the same behavior as the GRU model in this task.

9.2.2. Extrapolation

Figures 9.7 and 9.8, and Table 9.4 present the results achieved by the LSTM model in the extrapolation task. The LSTM model demonstrated a larger underestimation for lower values and a larger overestimation for higher values for the 'job_start' parameter compared to the GRU model, which can be seen in the Figure 9.7 for this parameter. Distribution in

Table 9.3.: LSTM: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.0045	0.06706	0.04573	0.9955
job_end	0.00941	0.09701	0.06375	0.99059
compute_time	0.00615	0.07841	0.0421	0.99385
input_files_transfer_time	0.36813	0.60674	0.28991	0.63187
output_files_transfer_time	0.2742	0.52364	0.19949	0.7258

Figure 9.8 has a higher peak at -1 and a longer tail between 2 and 3. Predictions for other parameters mirror the GRU model predictions.

Table 9.4.: LSTM: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.07245	0.26916	0.20505	0.92755
job_end	0.16842	0.41039	0.2798	0.83158
compute_time	0.01136	0.10658	0.06104	0.98864
input_files_transfer_time	1.14064	1.06801	0.39858	-0.14064
output_files_transfer_time	1.07033	1.03457	0.4086	-0.07033

9.3. Transformer

The best Transformer model used a batch size of 128, a single encoder layer with a hidden size of 16, four attention heads, and a window size of 100 without overlap.

9.3.1. Interpolation

Figures 9.9 and 9.10, and Table 9.5 present the results achieved by the Transformer model in the interpolation task. As the GRU and LSTM models, the Transformer model predicted the 'job_start', 'job_end', and 'compute_time' parameters with very high accuracy, with an R-score near 0.99. The predictions for 'input_files_transfer_time' and output_files_transfer_time have some errors, which is evident from Figure 9.9.

9.3.2. Extrapolation

Figures 9.11 and 9.12, and Table 9.6 present the results achieved by the Transformer model in the extrapolation task. In this task, the Transformer model performed well according

Table 9.5.: Transformer: Interpolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.0048	0.06926	0.04693	0.9952
job_end	0.00996	0.09979	0.06764	0.99004
compute_time	0.00704	0.08392	0.04788	0.99296
input_files_transfer_time	0.37241	0.61026	0.2947	0.62759
output_files_transfer_time	0.3073	0.55435	0.21206	0.6927

to metrics and not much worse than the GRU and LSTM models according to KDE plots in Figure 9.12. However, we can observe a systematic underestimation in a low range and a systematic 'job_start' and 'job_end' predictions. The variance in 'compute_time' predictions can be observed in Figures 9.11 and 9.12.

Table 9.6.: Transformer: Extrapolation accuracy metrics

Parameter	MSE	RMSE	MAE	R^2
job_start	0.07316	0.27047	0.20818	0.92684
job_end	0.20759	0.45562	0.3276	0.79241
compute_time	0.0387	0.19672	0.10659	0.9613
input_files_transfer_time	1.46237	1.20928	0.47813	-0.46237
output_files_transfer_time	1.21525	1.10238	0.45306	-0.21525

9.4. Discussion

This scenario contains tasks, that are harder than the tasks in previous scenarios. The job parameters have more complex distributions, and the platform configuration represents a sophisticated infrastructure.

The GRU and LSTM models showed almost identical accuracy in both tasks, and the Transformer model demonstrated worse performance in the extrapolation task. All models failed to fit the exact distributions of all parameters in the extrapolation task, except the 'compute_time' parameter.

The results of this scenario demonstrate, that all three model types can generate accurate predictions for 'job_start', 'job_end', and 'compute_time' parameters in interpolation tasks, but the behavior of 'input_files_transfer_time' and 'output_files_transfer_time' can not be accurately predicted because of very narrow value distributions with long tails. The accuracy plots demonstrate the bias in model predictions for these parameters. This behavior may be influenced by complex interactions within the platform, such as the existence of various routes from the node storing the dataset to the worker node that processes the job. Since information about the platform's configuration is not included in the model inputs, it cannot be utilized for making predictions.

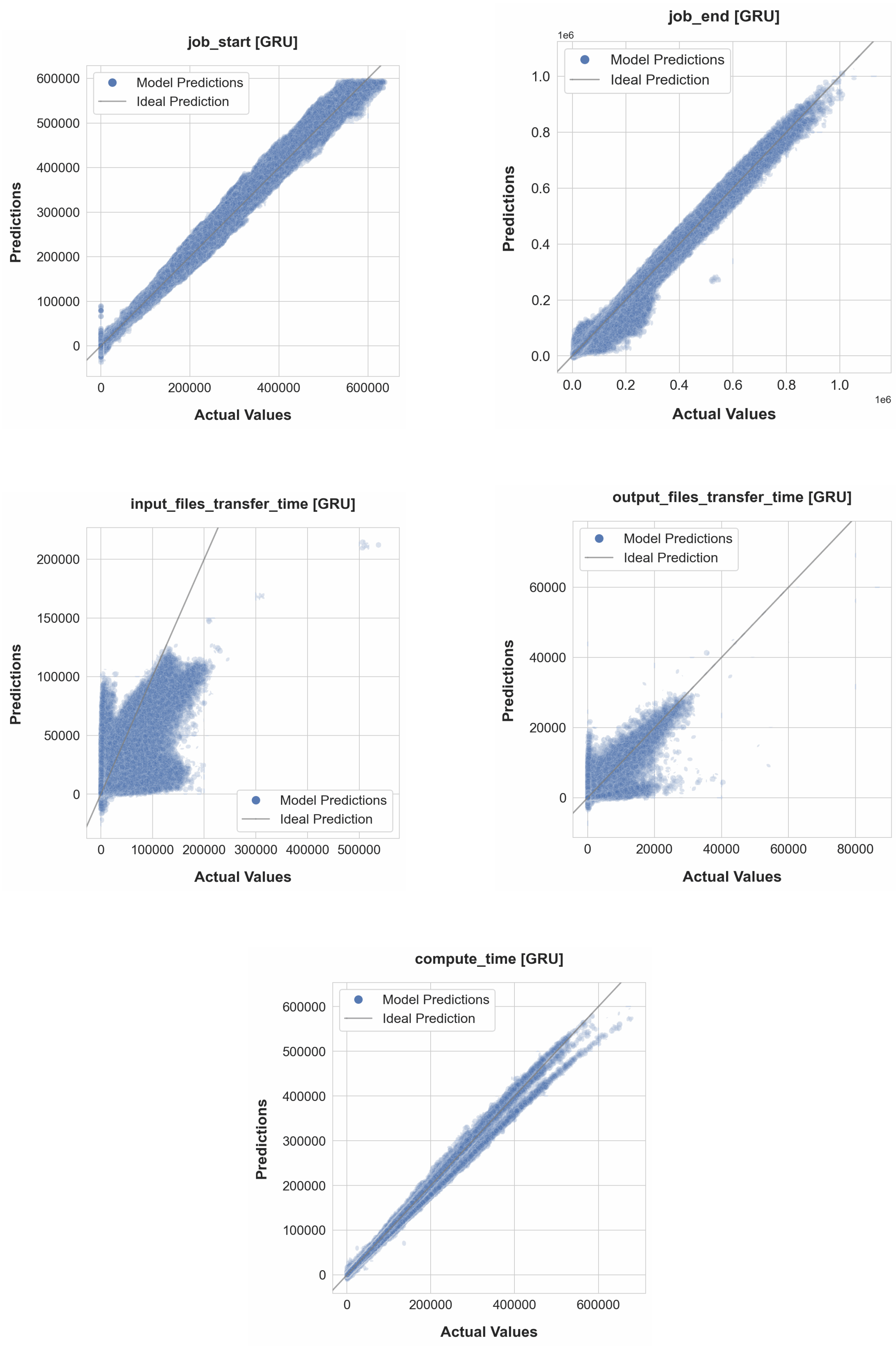


Figure 9.1.: GRU Interpolation Accuracy

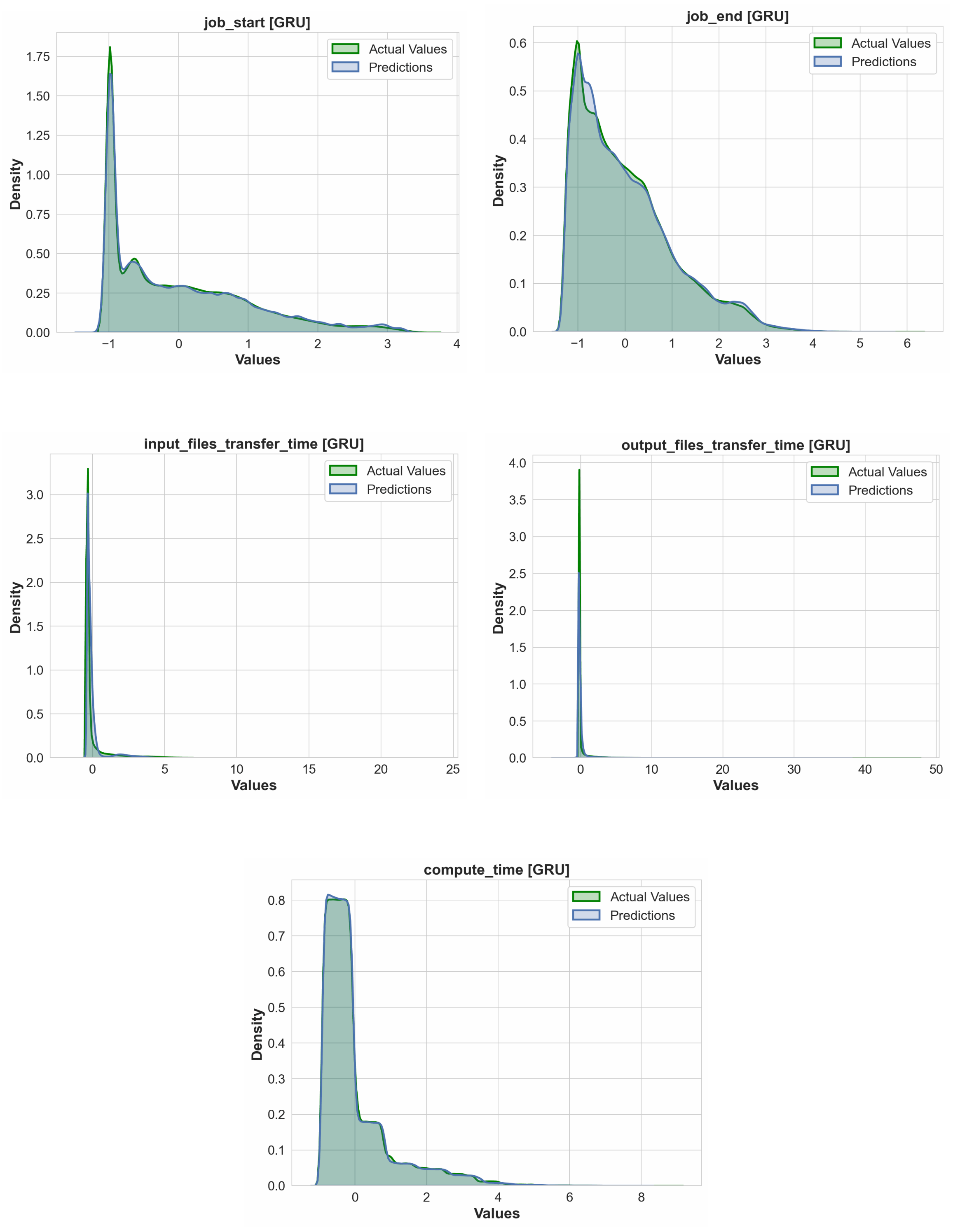


Figure 9.2.: GRU Interpolation KDE

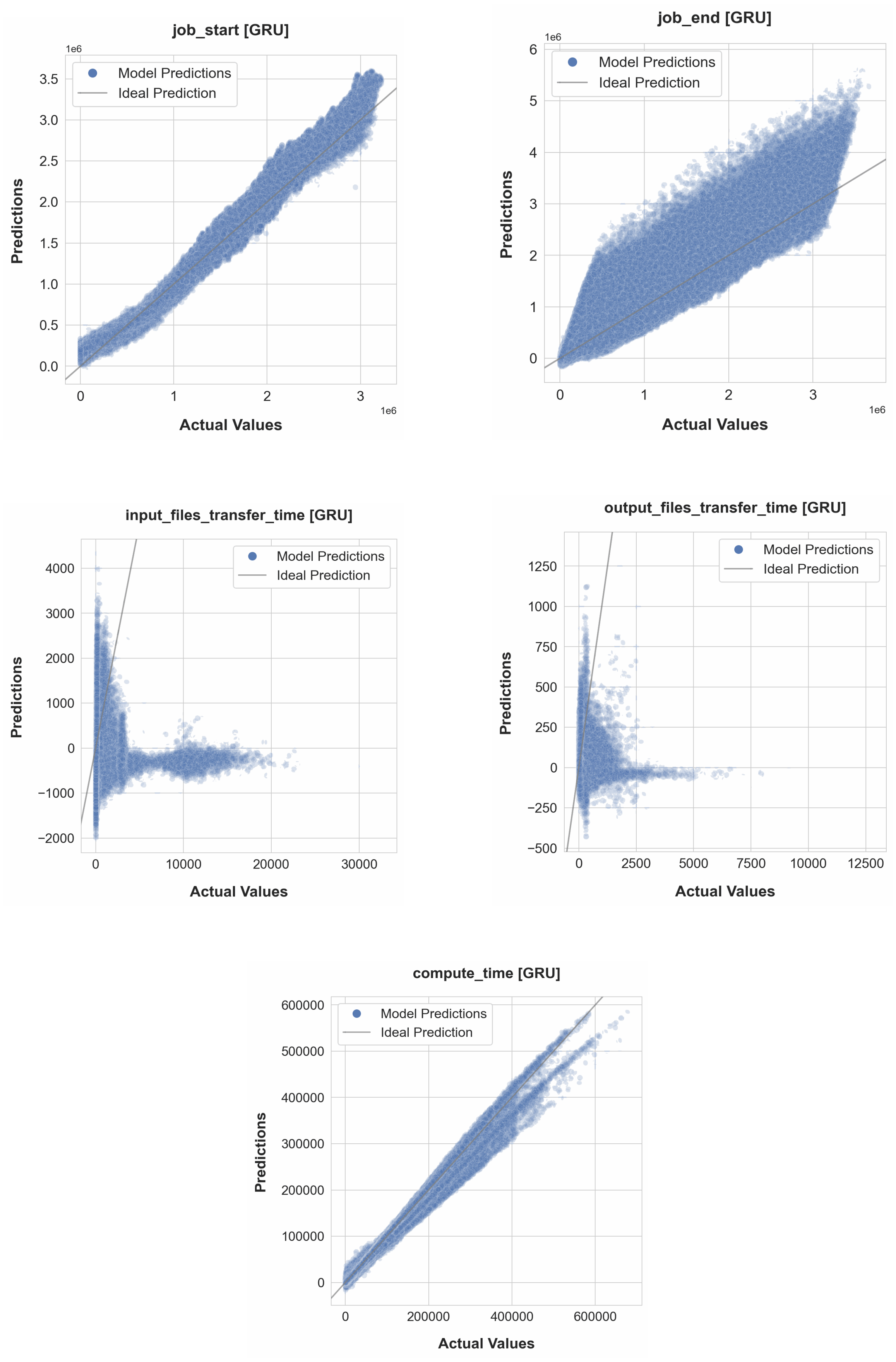


Figure 9.3.: GRU Extrapolation Accuracy

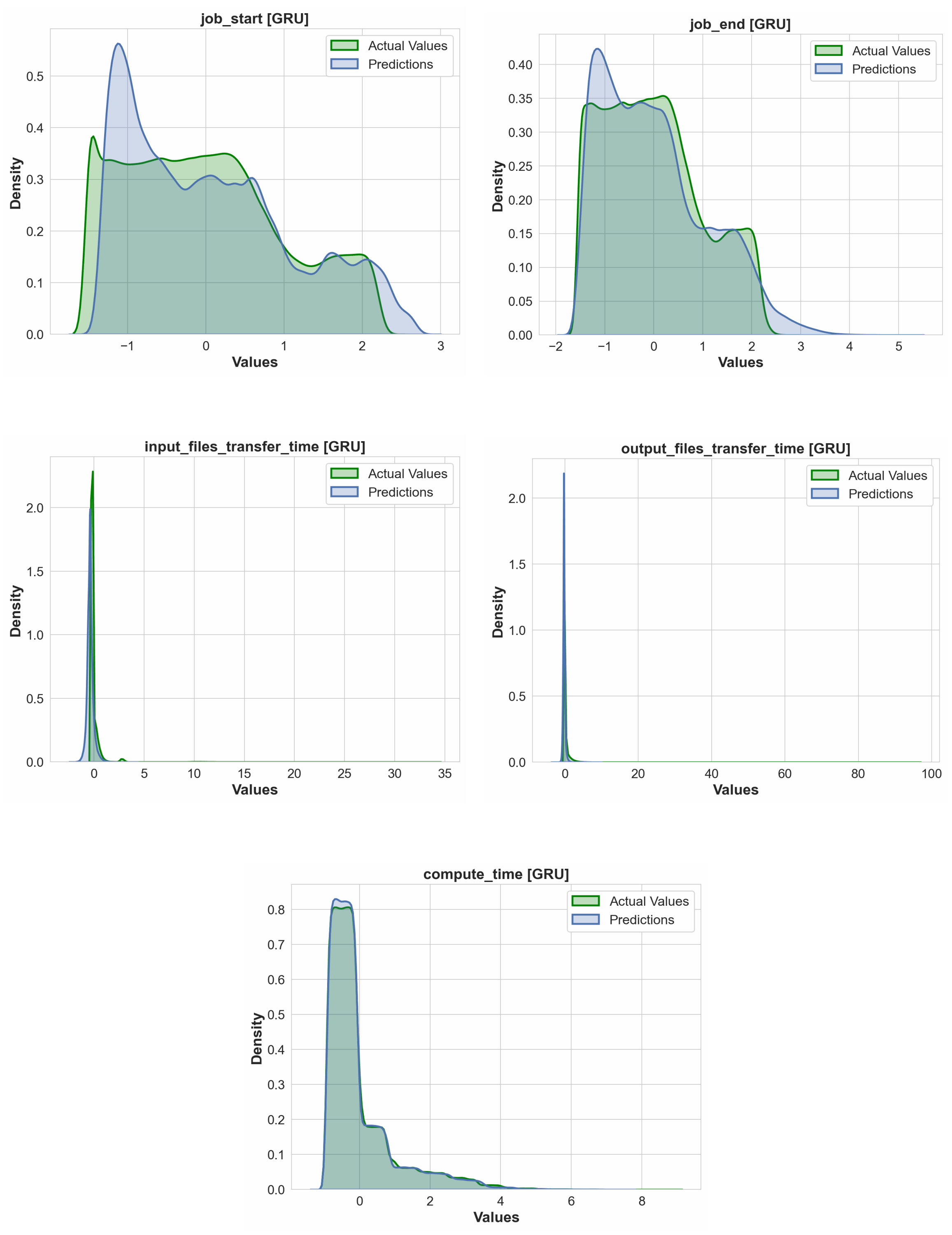


Figure 9.4.: GRU Extrapolation KDE

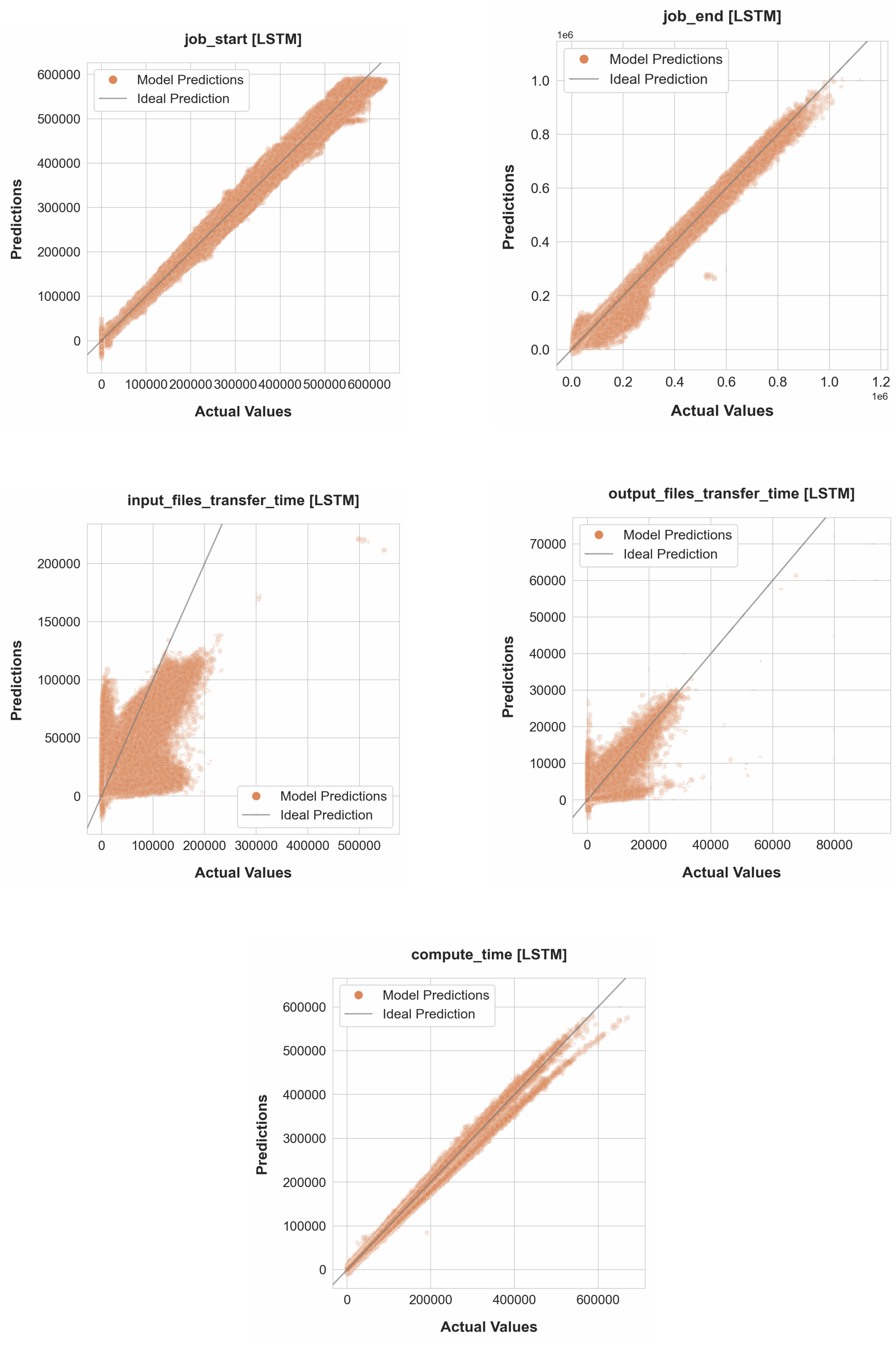


Figure 9.5.: LSTM Interpolation Accuracy

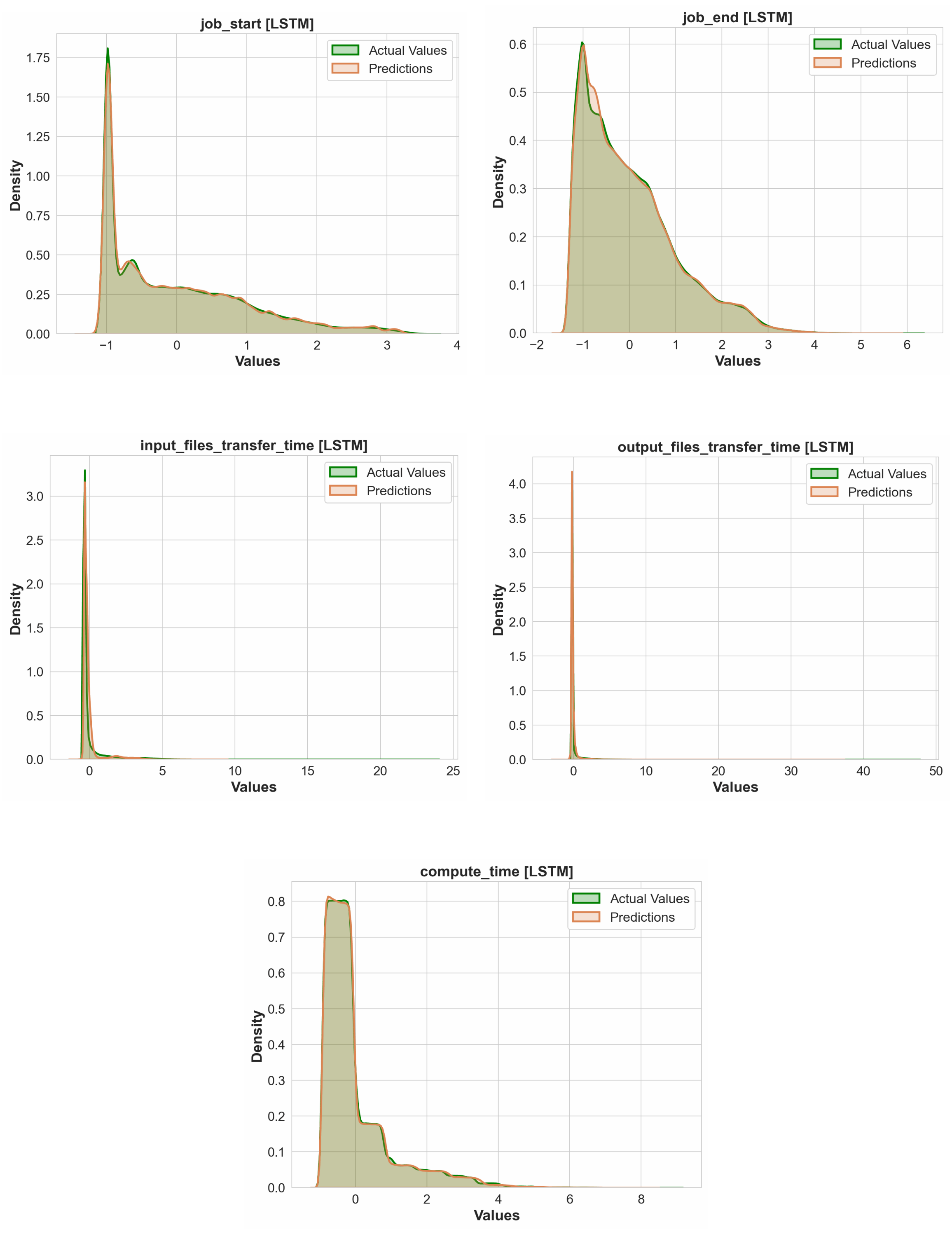


Figure 9.6.: LSTM Interpolation KDE

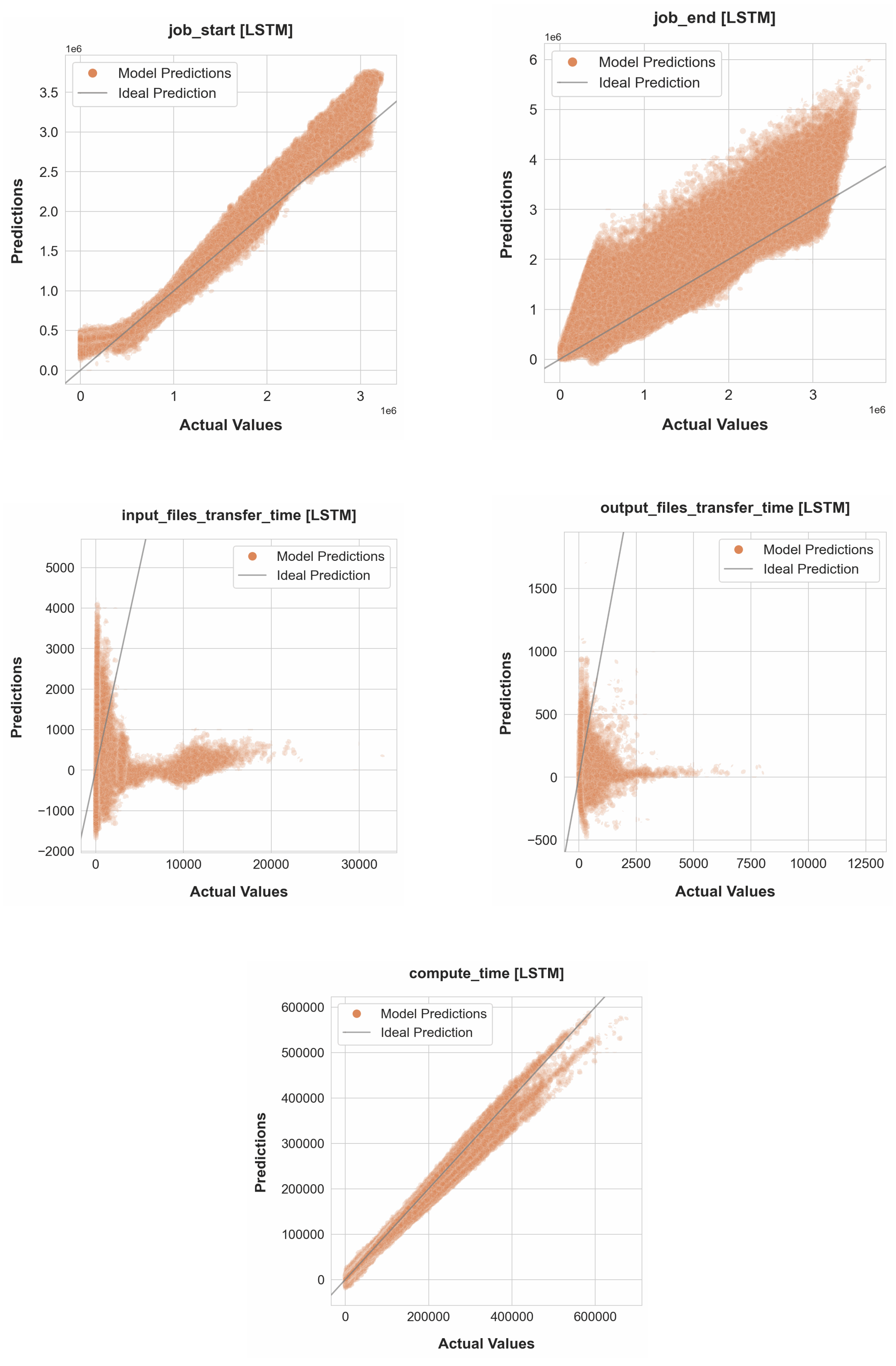


Figure 9.7.: LSTM Extrapolation Accuracy

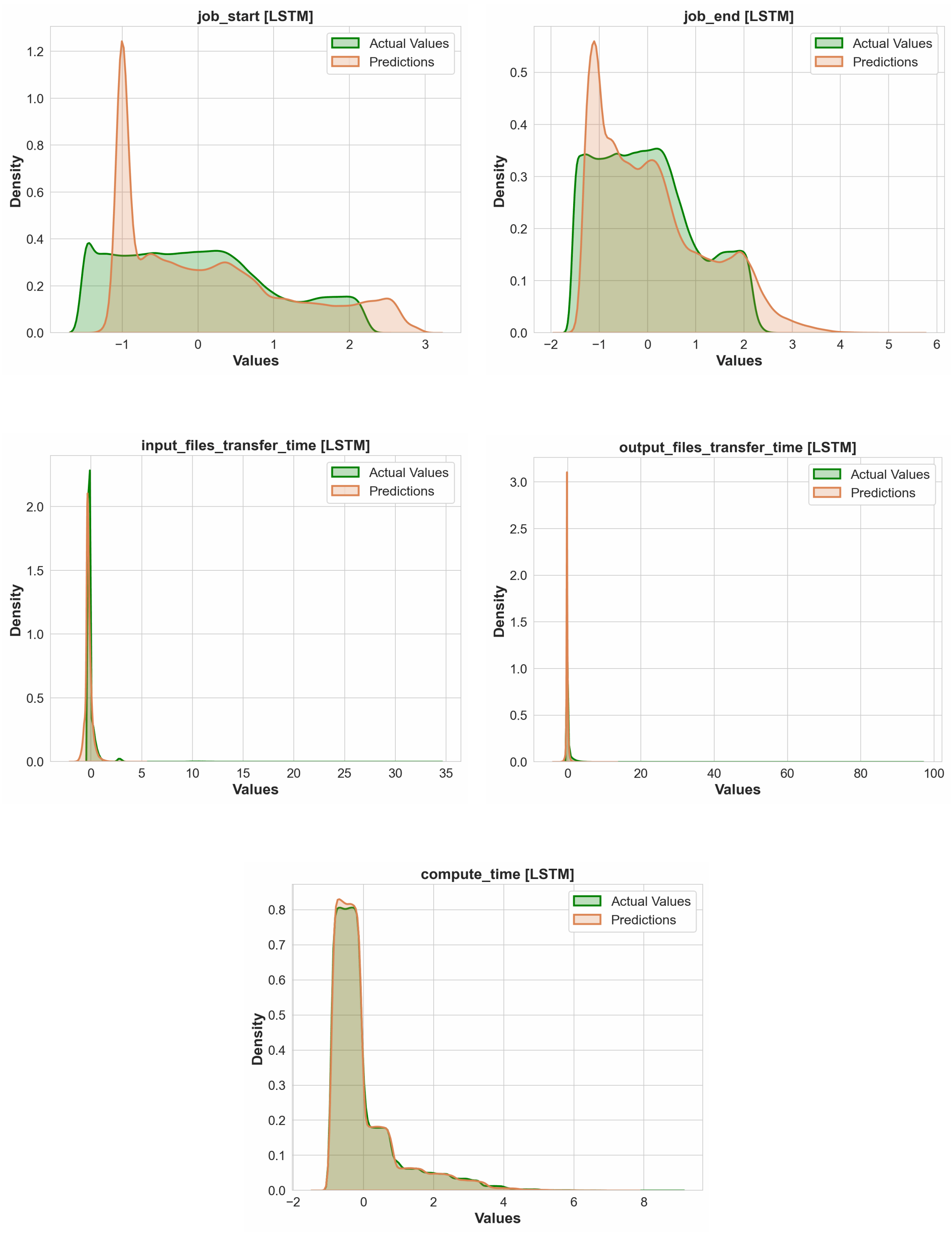


Figure 9.8.: LSTM Extrapolation KDE

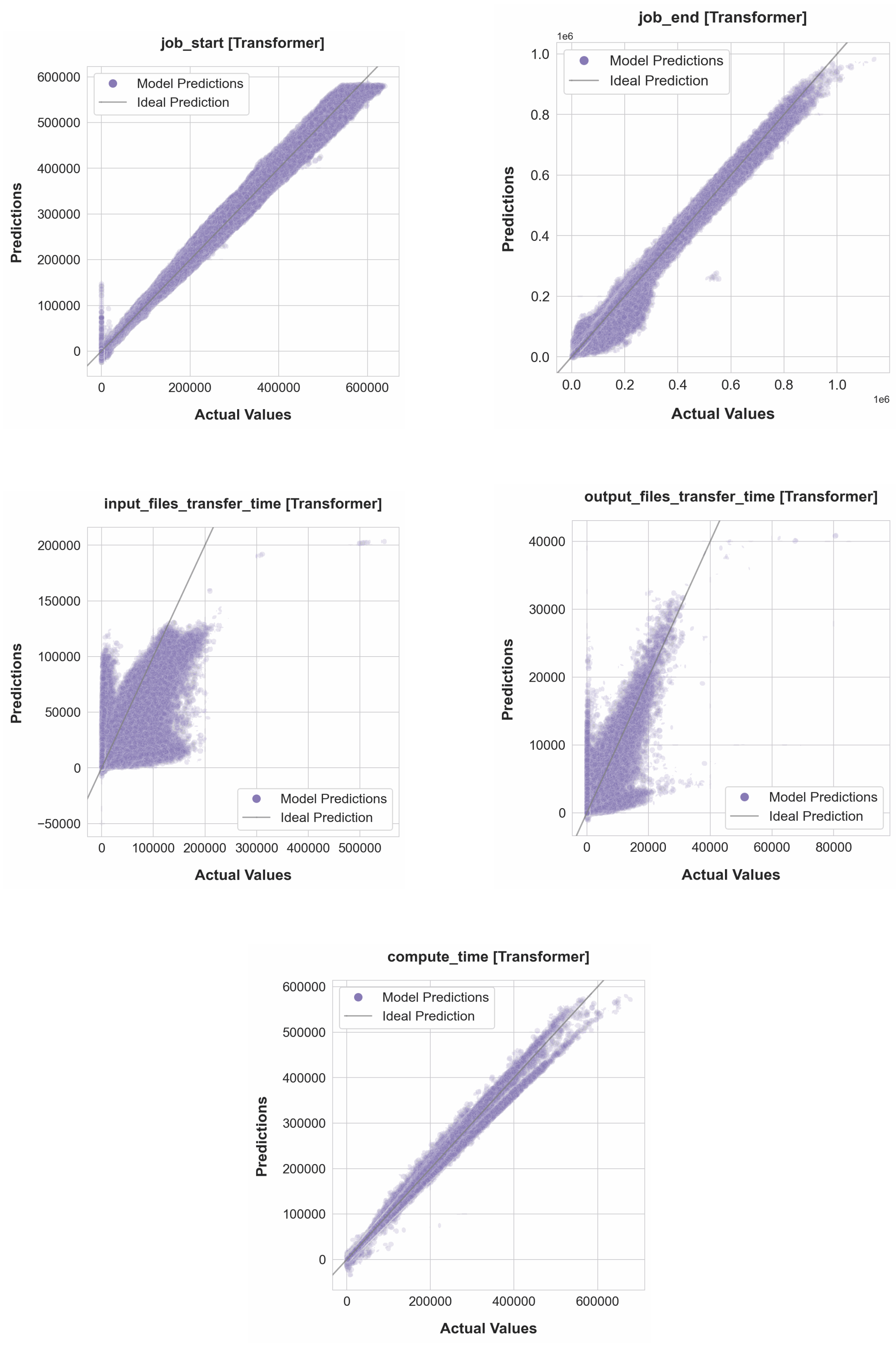


Figure 9.9.: Transformer Interpolation Accuracy

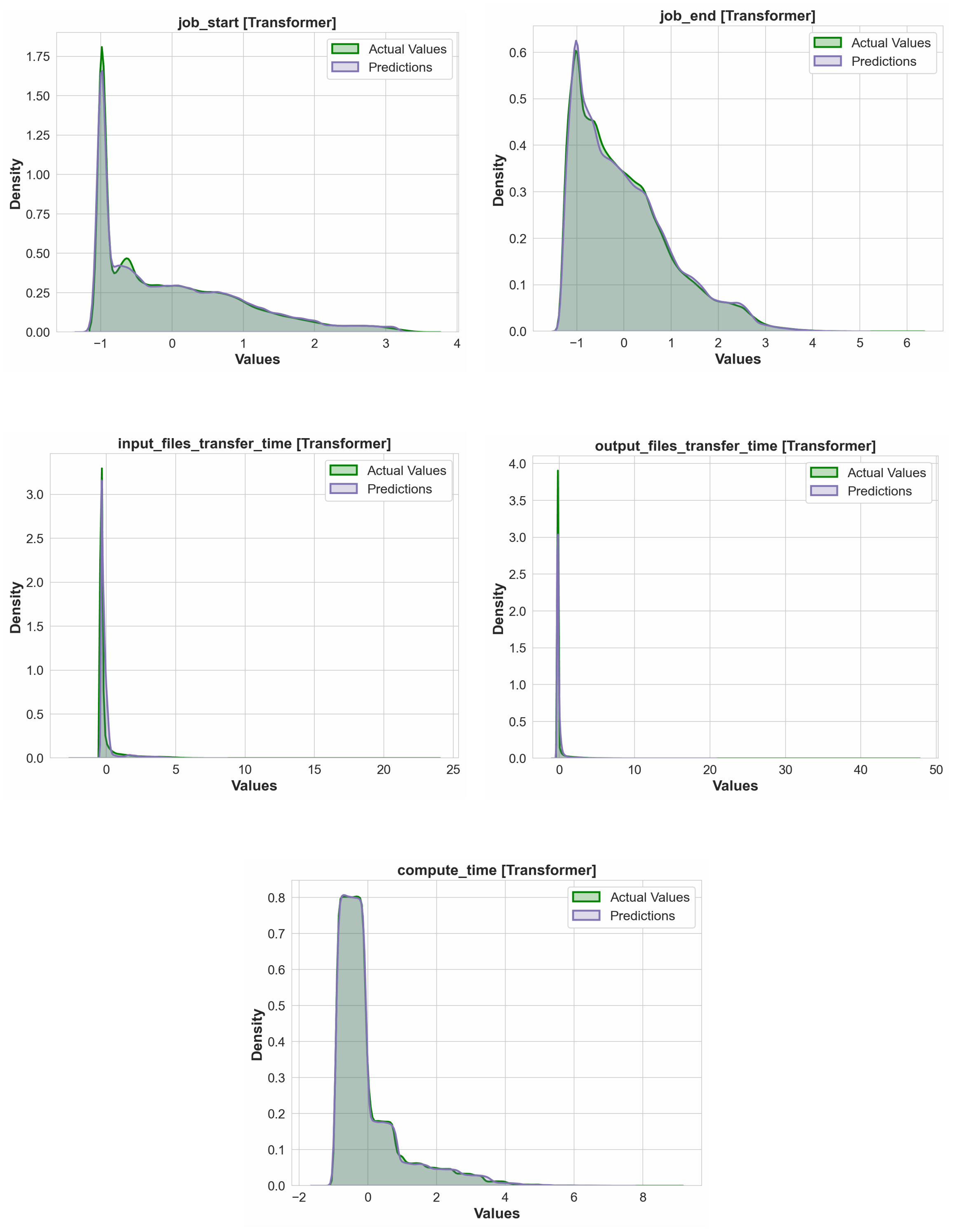


Figure 9.10.: Transformer Interpolation KDE

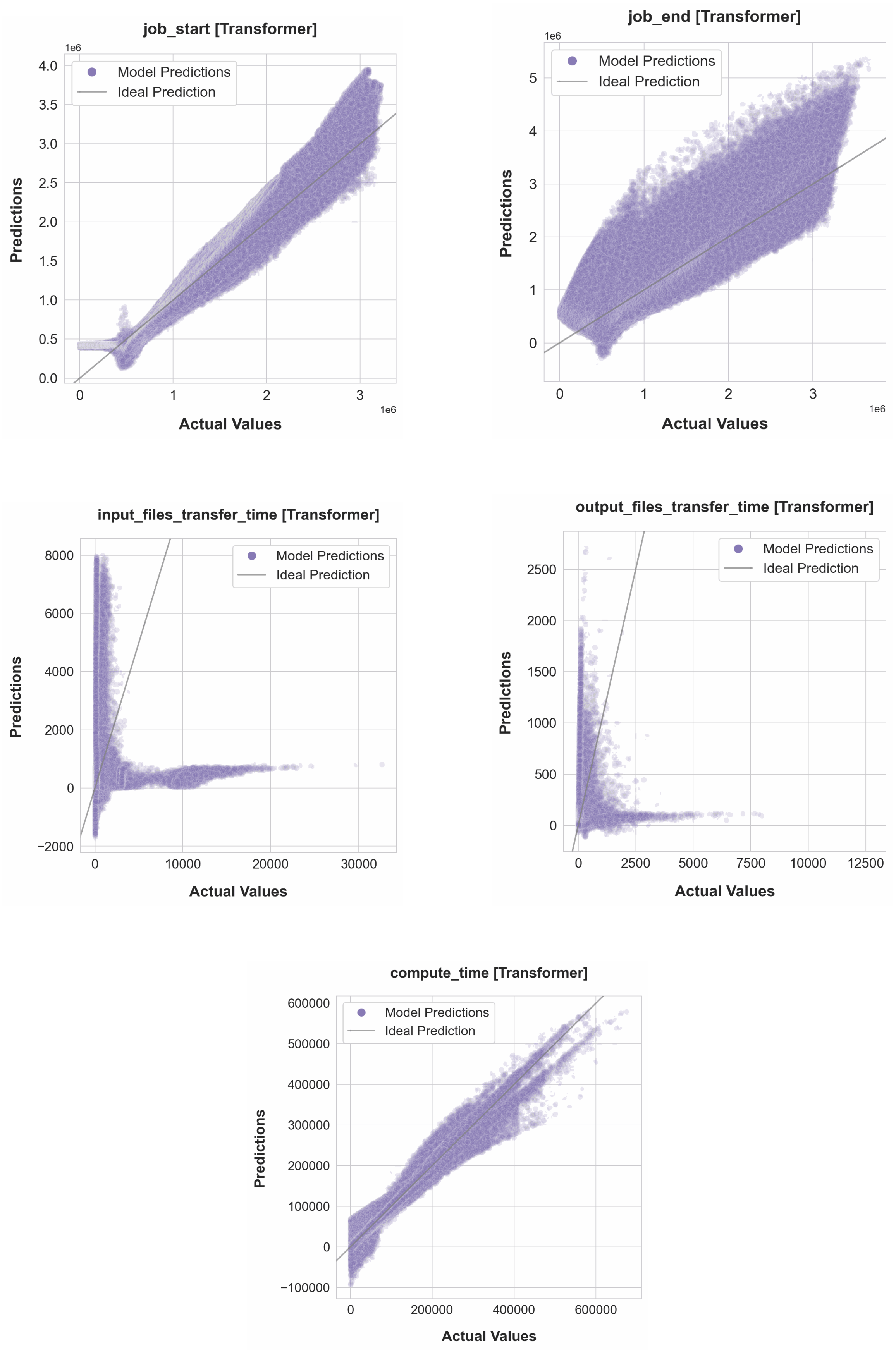


Figure 9.11.: Transformer Extrapolation Accuracy

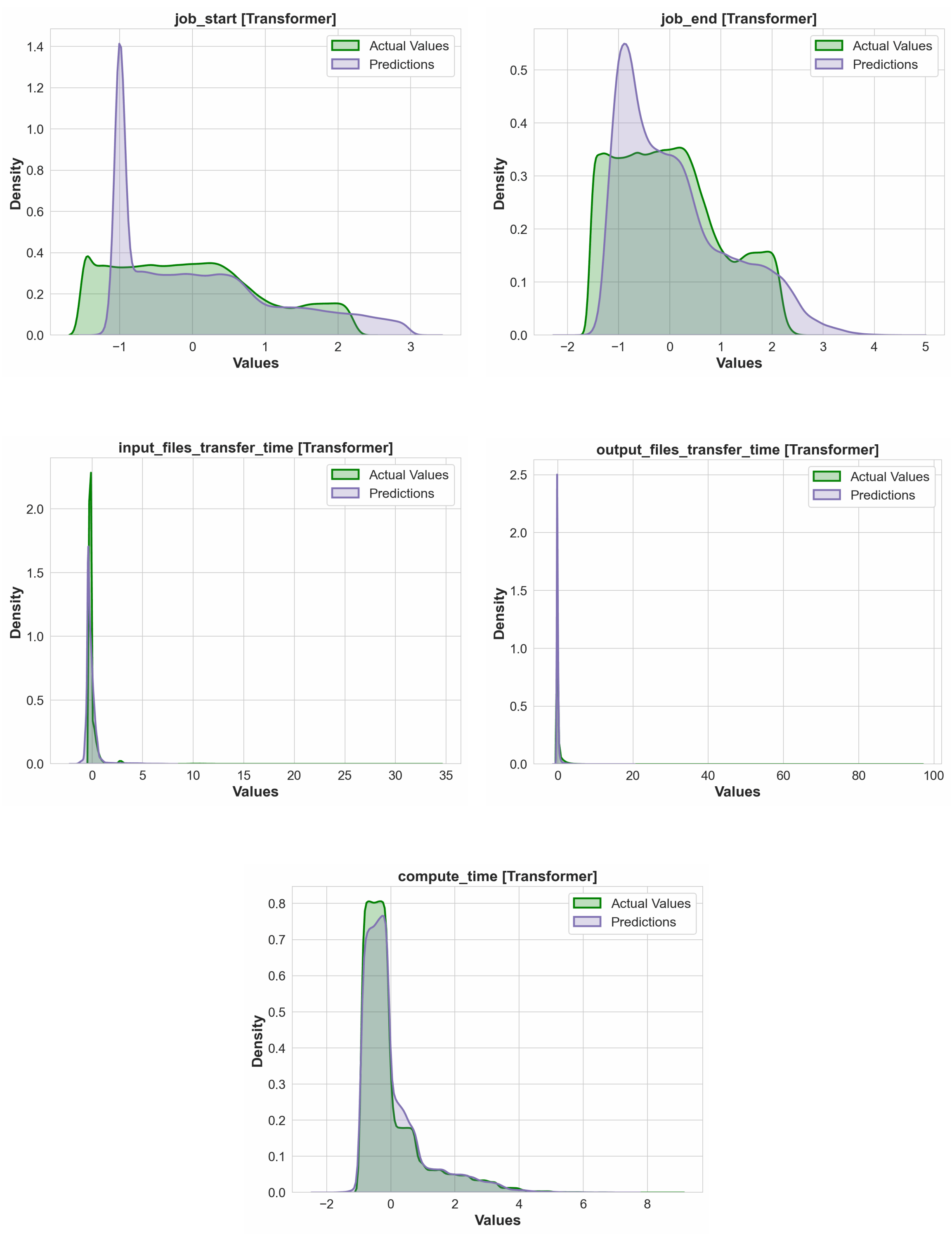


Figure 9.12.: Transformer Extrapolation KDE

10. Fourth Scenario: Platform Generalization

In previous scenarios, we trained and evaluated models for specific platforms. These models are generalizable across different workloads, but not generalizable across different platforms. In this scenario, we aim to add information about platform topology to our models, to enable the reuse of the pre-trained model for new platforms, eliminating the need to train a new model for each specific platform topology. The data preparation process for this scenario is described in Section 5.5.

10.1. Categorical Data

In previous scenarios, all input and output features were numerical. The numerical representation is not suitable for several platform features, presented in Listing 5.5 and Listing 5.6. Features 'node_index', 'node_type_index', 'link_index', 'src_node_index', and 'dst_node_index' despite being represented as numbers, actually function as categorical data because they are used as identifiers or to represent discrete entities. Categorical features can not be used directly in RNN and Transformer models, they need to be converted into a numerical format.

In our research, we evaluated two methods for representing categorical data within numerical models: one-hot encodings and embeddings [26]. One-hot encodings involve converting each categorical value into a binary vector, where only one element is '1' (indicating the presence of the category), and the rest are '0's. This method creates a distinct dimension for each category, making it straightforward but potentially leading to very high-dimensional data for features with many categories. Embeddings map each category of a single feature to a dense vector of continuous numbers, reducing the dimensionality and capturing more complex relationships between categories. This approach allows the model to learn an efficient numerical representation of the categorical data.

When the number of categories for each column can be very large, one-hot encodings can lead to sparse matrices with many dimensions, which can be computationally inefficient and difficult for the model to process [17]. Embeddings, by reducing dimensionality and capturing similarity between categories, offer a more scalable and informative approach for handling categorical data in such scenarios. Therefore, we employed the embeddings.

When creating an embedding for a numerical variable, the number of unique categories must be explicitly set to define the dimensions of the embedding matrix, ensuring each unique value is consistently represented by a distinct vector within the model's learned feature space. This size of the vocabulary for the embedding layer should be set to the number of unique categories in your categorical variable plus one for handling any unseen categories or to provide a padding index if needed.

10.2. Scaling

In previous scenarios, we employed data standardization, described in Section 4.1.2. This process is necessary for the correct prediction of numerical sequences. During the training phase, the scaler for each parameter is fitted on the whole dataset, prediction is made in the normalized space, and then the predictions are scaled back to the original scale.

When we employ no scaling for job parameters and platform parameters, then we get unstable models that give no meaningful predictions. For example, if we take a system with 10 nodes with 100 CPU cores in each node, the mean will be subtracted from each value and it will be divided by the standard deviation. All values of this feature are identical in this case, and therefore also the standard deviation is zero, which results in a division by zero during scaling process. This scenario leads to computational errors during model training and can render the model unable to learn or make any meaningful predictions (e.g. predicting a NaN for each parameter).

For another example involving some variance in the number of CPU cores, consider a system with 5 nodes having 1, 2, 3, 4, and 5 cores respectively. This platform, when scaled, will have the same scaled values for CPU cores as a system with 5 nodes having 10, 20, 30, 40, and 50 cores respectively, when a Standard Scaler is applied. The scaler might suppress the relative differences in computational capabilities between different nodes. For instance, a node with 50 cores is significantly more powerful than a node with 5 cores, but scaling may obscure this difference.

When we employ scaling for job parameters (as in the previous experiments) and no scaling for platform parameters (e.g. the number of CPU cores), we get predictions with a lot of noise. Figures 10.1 and 10.2 presents plots with predictions for 'job_start', 'job_end' and 'compute_time' parameters, predicted by the GRU model. The left plots represent the predictions made by the model from the third scenario, and the right plots represent the predictions made by the model that utilize additional platform information. Predictions for 'job_end' and 'compute_time' made by the model that uses the platform information contain many errors. These errors are not present in predictions of models without the platform information. The errors can be observed in the accuracy plots. However, the distribution is predicted mostly correct, which is evident from the KDE plots. The LSTM and Transformed models demonstrated identical behavior.

When we trained this model for a further 15 platforms, we got a poor prediction for the original platform. Figure 10.3 presents the plots with predictions, generated by a GRU model from the third scenario without the platform information (left plots) and the plots with predictions, generated by a GRU model trained on datasets from 15 different platforms including the platform information (right plots). In this case, the distributions of predicted parameters demonstrate large deviations from the actual distributions for all parameters. The evaluation dataset contains the same simulations in all cases. This behavior was observed in all GRU, LSTM, and Transformer models.

A possible workaround could be the scaling of platform parameters in the dataset which will cover many combinations of all possible platforms. Instead of training the model on a dataset with 15 platforms, we could perform simulations for thousands or even millions of different platforms, which is out of the scope of this work. Such a model training process is not feasible because of the data volume required for the training, compared to the training of specialized models for each platform.

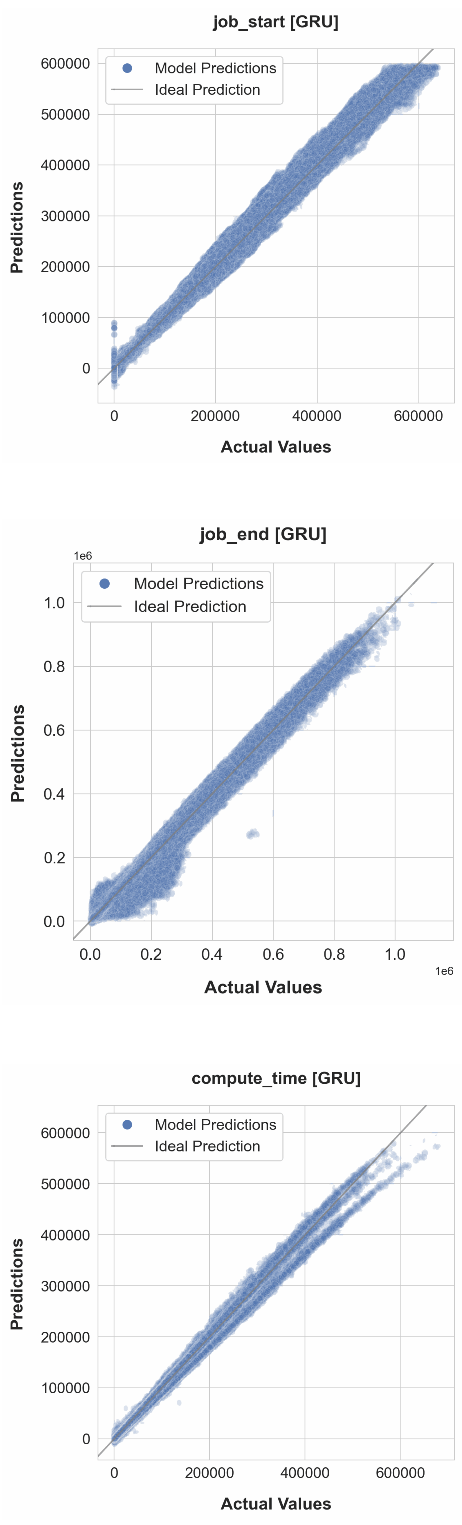
10.3. Discussion

The integration of numerical and categorical parameters in modeling is challenging, particularly when translating categorical platform characteristics into forms that are usable for numerical predictions. When the number of categories is unknown beforehand, the approach of setting a fixed size of vocabulary for an embedding layer is not applicable.

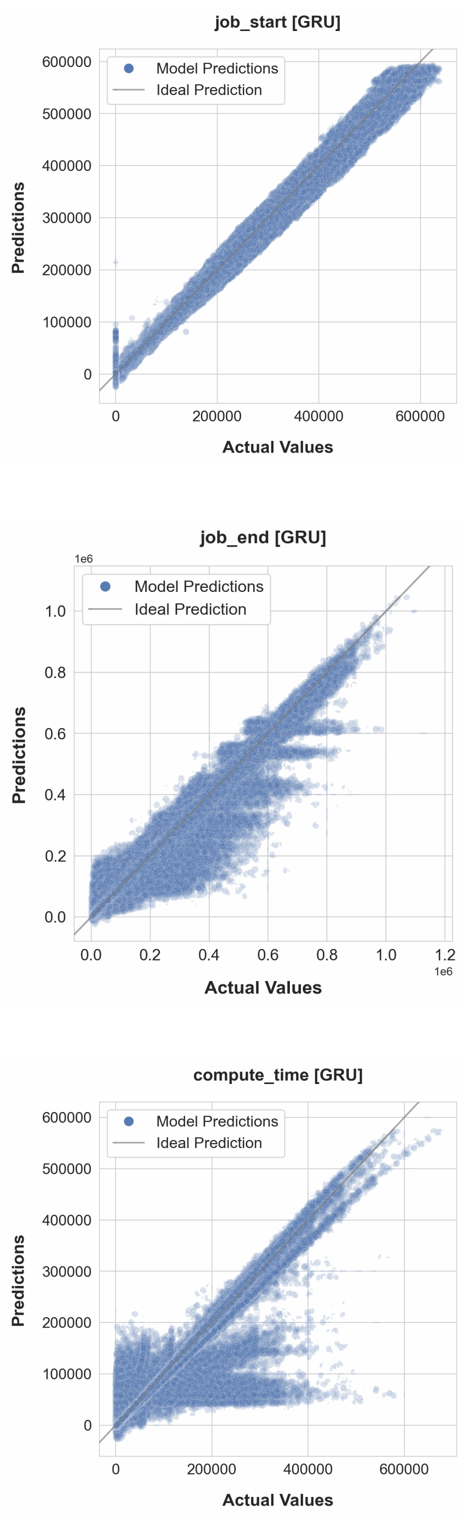
Models that used unscaled platform information underperformed compared to models, that did not use the platform information. The inclusion of unscaled parameters in the models can lead to instability, affecting the accuracy of the predictions.

Training models on data from multiple platforms can introduce complexities and variability that degrade performance when predicting parameters for a previously known single platform. This occurs because the model may overfit to the specific characteristics of the new platforms, thus losing its generalization ability for the original platform.

Given these challenges, training dedicated models for specific platform configurations remains the most effective approach. It necessitates using simulators to generate the training data for each platform. Consequently, the need to employ a simulator remains indispensable.

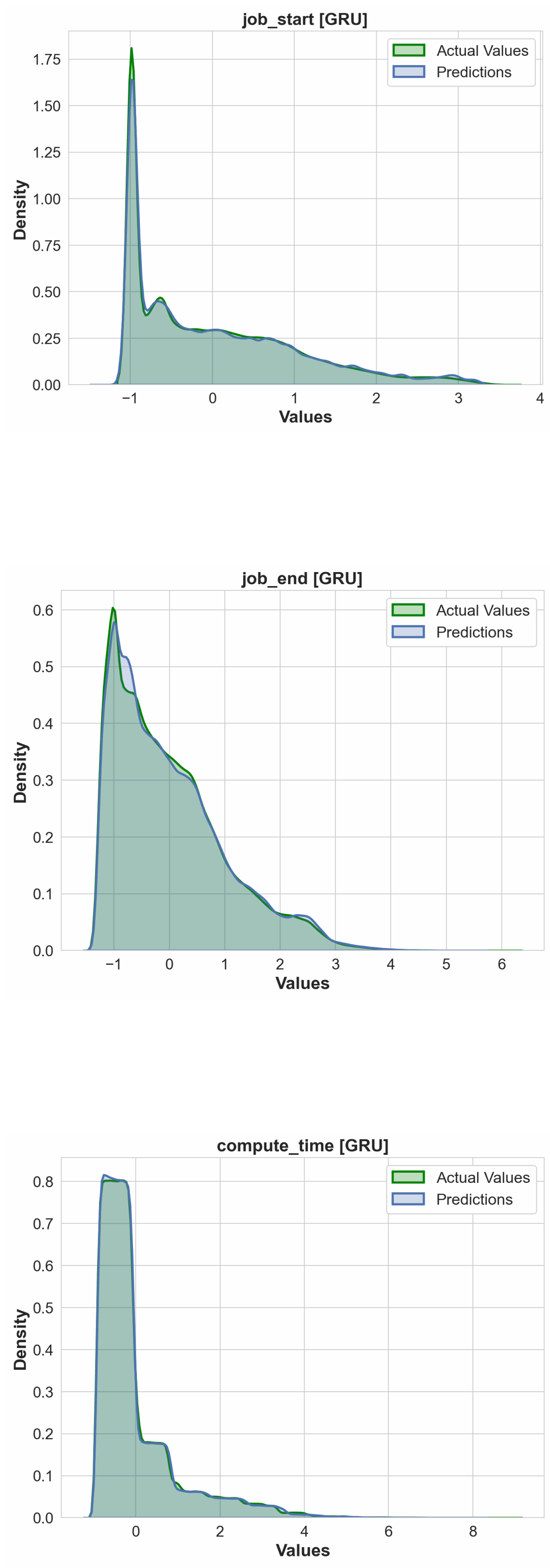


(a) Without platform information

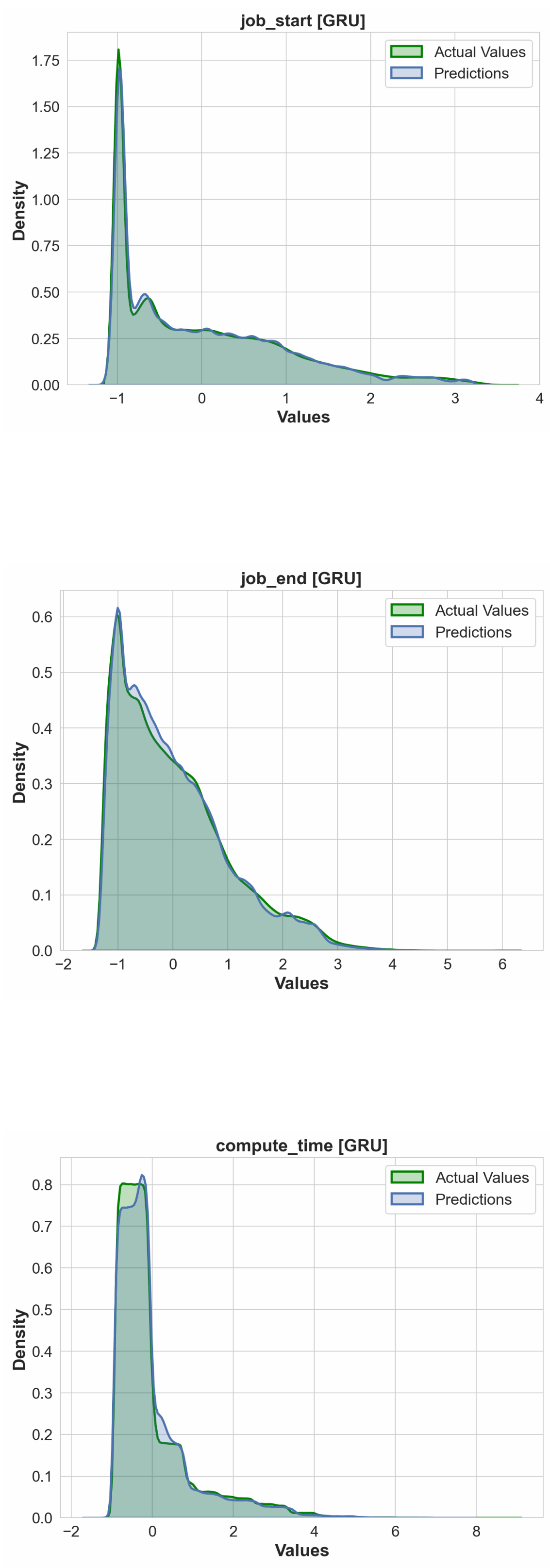


(b) With platform information

Figure 10.1.: GRU Interpolation Accuracy Plots: usage of platform information

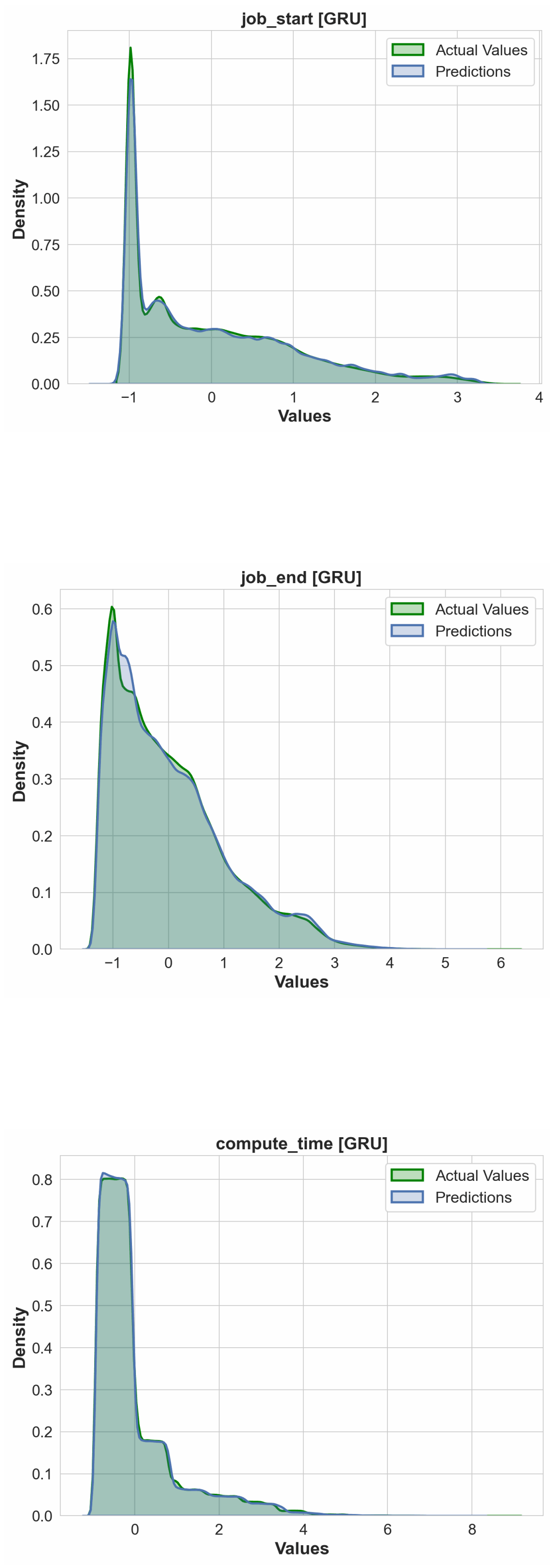


(a) Without platform information

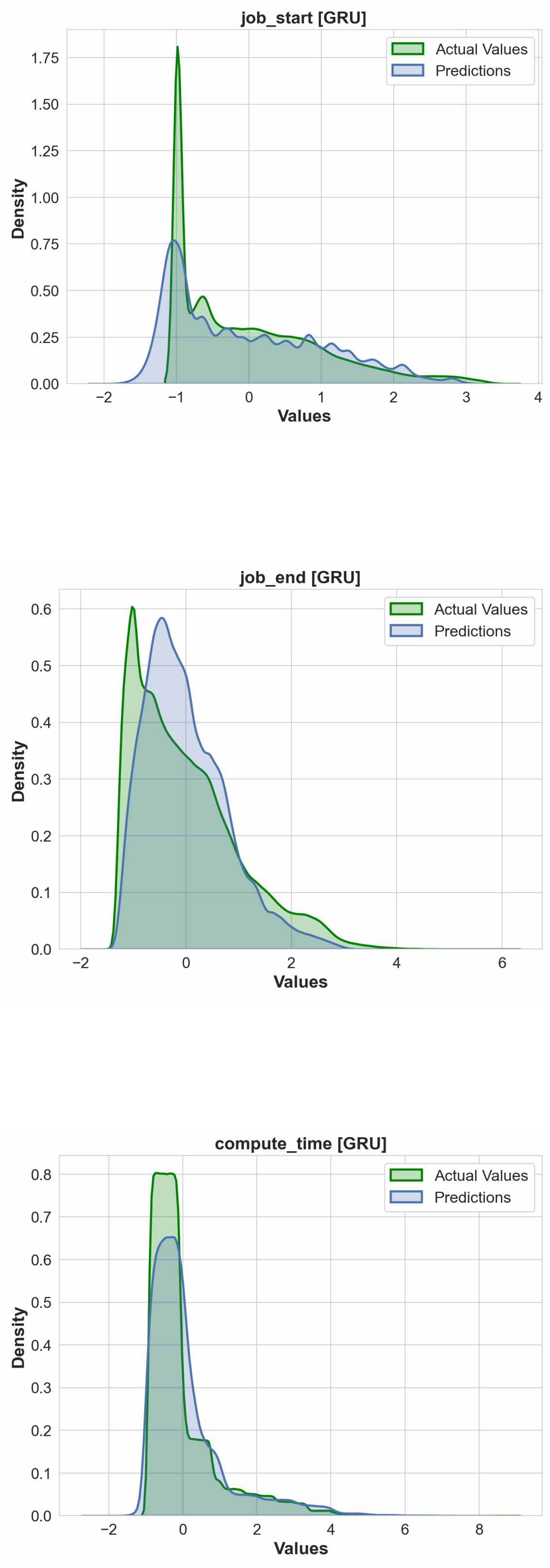


(b) With platform information

Figure 10.2.: GRU Interpolation KDE Plots: usage of platform information



(a) Without platform information



(b) With platform information for 15 platforms

Figure 10.3.: GRU Interpolation KDE Plots: 15 additional platforms

11. Evaluation and Discussion

In previous chapters, we showed that our method is capable of accurately predicting simulation results for specific platforms. In this chapter, we assess the amount of data required to effectively train the model, determine the optimal number of training epochs for best results, compare the influence of hidden size on model training, identify which input parameters significantly impact specific output parameters, and evaluate the time taken by the model for inference across varying sequence lengths.

11.1. GQM Plan

Following GQM Plan [4] is used for validation of this work.

11.1.1. Effectiveness in Predicting Job Metadata for Fixed Platforms

Goal 1: Assess the effectiveness of deep neural network models for predicting job execution metadata in distributed computing simulations on fixed platforms.

Question 1.1: How accurately do the models predict job execution metadata across different simulation scenarios?

- Metric 1.1.1: Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared between predicted and actual simulation results for each model type across various scenarios.
- Metric 1.1.2: Accuracy plots with point wise comparison of predicted values and actual values.
- Metric 1.1.3: KDE plots with comparison of distributions of predicted values and actual values.

Question 1.2: What is the optimal amount of training data required to achieve reliable prediction accuracy?

- Metric 1.2.1: Analysis of model accuracy as a function of training dataset size.
- Metric 1.2.2: Identification of the point of diminishing returns for accuracy as training data increases.

Question 1.3: How do the models perform in terms of computational efficiency and training dynamics?

- Metric 1.3.1: Training time per epoch for each model type.
- Metric 1.3.2: Total training duration and computational resources consumed by each model type.
- Metric 1.3.3: Number of epochs required to reach convergence for each model type.
- Metric 1.3.4: Inference time for processing sequences of 10,000 jobs.

Question 1.4: Can the models extrapolate to predict job batches of sizes not encountered during training?

- Metric 1.4.1: Accuracy of predictions for job batch sizes beyond those seen during training.
- Metric 1.4.2: Comparative analysis of extrapolation errors across different model types.

11.1.2. Platform Generalization

Goal 2: Investigate the adaptability of performance models to variable platform configurations in distributed computing simulations.

Question 2.1: How does the inclusion of platform variability affect the generalization ability of performance models?

- Metric 2.1.1: Comparison of model accuracy on known versus novel platform configurations.
- Metric 2.1.2: Effectiveness of models in adapting to changes in platform configurations without retraining.

11.2. Optimal Dataset Size for Model Training

We aim to identify the smallest dataset, required for model training to predict the data from the third scenario, described in Section 5.4. For evaluation, we use hyperparameters of the best models from the third scenario, described in Chapter 9. Each training dataset contains the sequences with 5, 10, 20, 50, 100, 250, 500, 1.000, 1.500, and 2.000 jobs. We created training datasets with 1, 10, 20, 30, 40, 50, 75, 100, 150, 200, 300, 400, and 500 sequences of each length. For model evaluation, we use a dataset with 100 sequences of each length, which are not present in training datasets. All models are trained and evaluated on the same datasets.

11.2.1. Results

We employ the KDE plots for model comparison. Parameters pair 'job_start' and 'job_end' and pair 'input_files_transfer_time' and 'output_files_transfer_time' exhibited similar behavior.

We show the prediction for the parameter 'job_start' in Figure 11.4 for the GRU model, in Figure 11.6 for LSTM, and in Figure 11.8 for Transformer, and the prediction for the 'input_files_transfer_time' parameter in Figure 11.5 for the GRU model, in Figure 11.7 for LSTM, and in Figure 11.9 for Transformer. Comparison for prediction of the 'compute_time' parameter for all three models is presented in Figure 11.10. Figure 11.11 presents the plot for models trained on larger datasets.

Plots with models, trained on one sequence, show underestimation in predictions. With the increase in dataset size, the predictions improve.

11.2.2. Discussion

All three models give an adequate prediction for small datasets, containing 20-40 sequences of each length, and improve the prediction quality for larger datasets. The usage of 50 sequences of each length for model training is the optimal choice for all models in this use case, further increase of the dataset leads to marginal improvements. Predictions for more complex jobs or more complex platforms could require more training data.

In all evaluation scenarios, we use datasets, that are much larger than the optimal size. The models trained with data from the third scenario using 500 sequences of each length do not perform better than the models trained using 500 sequences of each length.

11.3. Training Epoch Duration

In this study, we aim to examine the relationship between the size of the dataset and the average duration of training epochs for each model, specifically focusing on how the increase in dataset size affects the time required to complete an epoch. The measurements were conducted on the system with RTX 3080 and i7-13700K using PyTorch 2.1.1+cu121.

11.3.1. Measurement Results

We trained 10 dedicated models of each type for each dataset for 300 epochs and calculated the median epoch time. All models have the same hyperparameters. The median training epoch duration for each model is presented in Table 11.1. The dataset with one sequence of each length contains results of simulations, containing 5, 10, 20, 50, 100, 250, 500, 1.000, 1.500, and 2.000 jobs (5.435 jobs in total). All time measurements presented in the table are expressed in seconds. The dataset with 100 sequences of each length contains 1.000 sequences. The data from this table is visualized in Figure 11.1.

Table 11.1.: Relationship between Dataset Size and Epoch Duration in seconds

Sequences of each length	**GRU**	**LSTM**	**Transformer**
1	0.0139	0.0142	0.0190
10	0.0162	0.0197	0.0262
20	0.0309	0.0332	0.0501
30	0.0442	0.0469	0.0666
40	0.0571	0.0633	0.0896
50	0.0690	0.0792	0.1109
75	0.1149	0.1209	0.1655
100	0.1813	0.1832	0.2510
150	0.2325	0.2751	0.4030
200	0.3228	0.3748	0.4952
300	0.4568	0.5131	0.7100
400	0.5994	0.6613	0.9303
500	0.7157	0.8556	1.1322

11.3.2. Discussion

We can observe a linear growth of mean epoch time with very small slope (near 1/400 for Transformer) in Figure 11.1. The GRU model exhibits the shortest training times among the evaluated models, which aligns with expectations given its simpler architecture compared to the LSTM and Transformer models.

The duration of each training epoch is very small - nearly 1 second for the largest evaluated dataset. That means, that even the model that requires hundreds or thousands of epochs can be trained in less than one hour on a similar system.

Each job contains 9 floating-point parameters and one integer parameter, which require 38 bytes for each job. The largest dataset with 500 jobs of each length contains 5000 simulations and in total the data for $5000 * 5435 = 27.175.000$ jobs. Taking the job size into account, we get the $27.175.000 * 38 = 1.032.650.000$ bytes, which is less than one gibibyte of data. Such volume can be moved to the GPU in a fraction of a second.

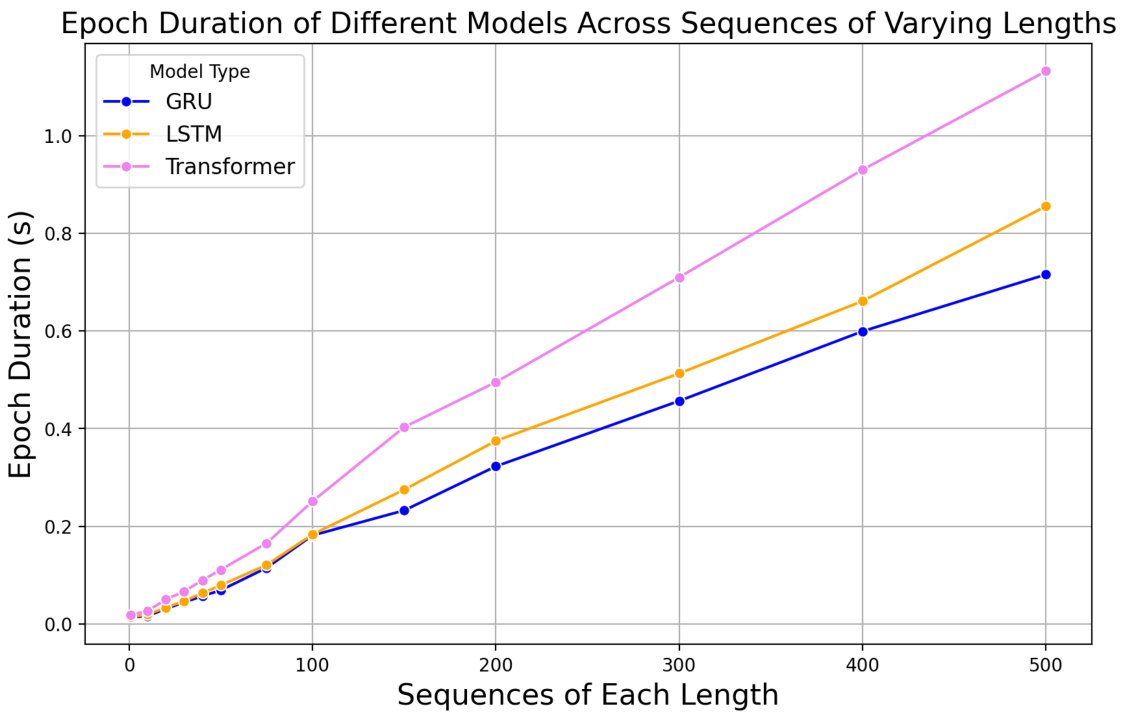


Figure 11.1.: Relationship between Number of Sequences in Dataset and Epoch Duration

11.4. Optimal Epoch Number for Model Training

We aim to identify the optimal number of epochs for model training to predict the simulation from the third scenario. To achieve this, we analyze the median training loss across epochs for each model type.

To make the losses comparable, we use the models with the same window and batch parameters and use the same training and evaluation datasets. All models utilize the same loss function (MSE Loss), the same optimizer and same weights initialization.

11.4.1. Results

We trained 10 models of each type using the same dataset, which contains 50 simulations of each length, and saved the training loss for each epoch. Plots of all loss sequences for each model type are presented in Figure 11.2. The broad blue area represents all values from 10 loss sequences of the GRU model, and blue circles represent the median value for specific epochs across the values from all 10 models.

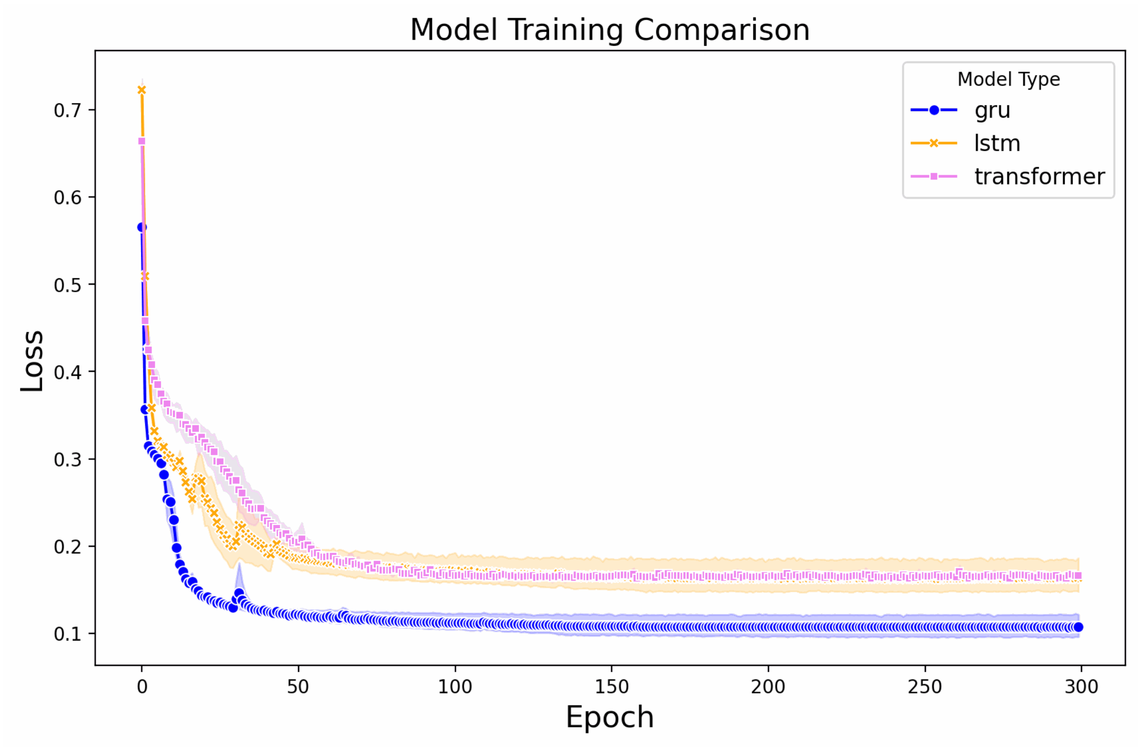


Figure 11.2.: Comparison of Training Rate

11.4.2. Discussion

The LSTM and Transformer models converged to the same loss, the LSTM models converged slightly faster than the Transformer, which is evident from Figure 11.2. Interestingly, all GRU models converged to a smaller loss value. LSTM models reached the plateau after 70 epochs, Transformer models reached the plateau after 80 epochs, and GRU models reached the plateau after only 50 epochs.

11.5. Model Inference Time

In our study, we quantify the duration required by each type of model to generate predictions for individual sequences. Furthermore, we measured the time dedicated to data preparation for each sequence, which encompasses the processes of reading the dataset from a file and performing data standardization.

11.5.1. Measurement Results

We trained the 10 models of each type with the same hyperparameters and with the same training data and applied them to the same evaluation datasets. The median time measurements for processing of single sequences by different model types are presented in Table 11.2. The data preparation process is identical for all three models, and the time of data preparation represents the time between the start of file reading and the end of loader initialization. Model inference time represents the time between the start of writing the first window into GPU memory and the end of transforming the predicted windows to a single n-dimensional array of predictions. All time measurements presented in the table are expressed in seconds. Figure 11.3 visualizes the relationship between sequence length and preparation time required for inference.

Table 11.2.: Data Preparation and Model Inference Time in seconds

Sequence Length	Preparation	GRU	LSTM	Transformer
5	0.01302	0.01151	0.00200	0.02633
10	0.01043	0.00200	0.00200	0.00200
20	0.00951	0.00099	0.00100	0.00099
50	0.00950	0.00100	0.00101	0.00200
100	0.00908	0.00100	0.00200	0.00101
250	0.01103	0.00402	0.00101	0.00250
500	0.01051	0.00151	0.00100	0.00100
1.000	0.01522	0.00204	0.00205	0.00099
1.500	0.01899	0.01671	0.00300	0.01016
2.000	0.01702	0.00200	0.00200	0.00100
10.000	0.04477	0.00701	0.00802	0.00201

11.5.2. Discussion

The dataset preparation time is higher than the model inference time and grows linearly with the sequence length. Simulations with 5, 10, 50, and 100 have the same dataset preparation time because all windows are zero-padded to the length of 100, which can be observed in Figure 11.3.

Each time we started the model inference process, we got a new distribution of values. Even the longest single sequence requires very little memory - a sequence with 10.000 elements contains only 90.000 floating point values and 10.000 integer values, which requires approximately 300KB of memory. This variability can be attributed to the inherent unpredictability of GPU memory allocation and retrieval processes on such a small scale.

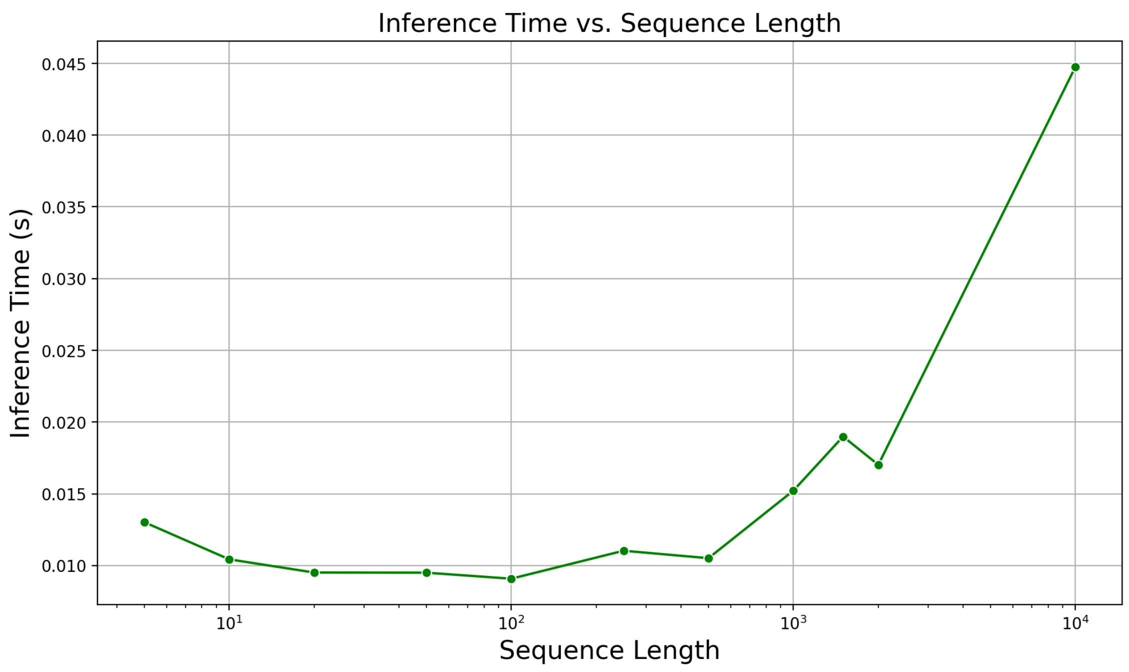


Figure 11.3.: Relationship between Sequence Length and Preparation Time

11.6. Job Parameters Influence

We employed the Feature Ablation [51] to measure the influence of the input features on predictions of each job parameter. Feature Ablation is a technique used to assess the importance of input features to a model's predictions by systematically removing or masking of individual features and observing the resulting impact on model performance. In the evaluation, we removed the input features, changed the window size, and observed the changes in prediction accuracy.

Parameters 'job_start', and 'job_end' are dependent primarily on the 'index' of the specific job, but are influenced by all parameters of all jobs in the window. Parameter 'compute_time' is not dependent on the job index and is influenced mostly by 'flops' parameter. Parameter 'input_files_transfer_time' is influenced mostly by 'input_files_size', and 'output_files_transfer_time', is influenced mostly by 'output_files_size'.

The behavior of 'compute_time', 'input_files_size', and 'output_files_transfer_time' reflects domain-specific dependencies between job parameters used for modeling dynamics in computational environments.

11.7. Incorporation of One Longer Sequence into Dataset

In Chapter 9 we observed the capabilities of our models in extrapolation tasks. In this task, the models trained on sequences with lengths 5, 10, 20, 50, 100, 250, 500, 1.000, 1.500, and

2.000 were applied to predict the parameters of sequences with length 10.000. We aim to investigate the improvement in predictions for long sequences after incorporating one additional sequence with 10000 jobs into the training dataset.

11.7.1. Results

We use the best models from Chapter 9, and the extrapolation dataset with 100 sequences of length 10.000. At first, we trained the model on the dataset with 50 sequences of each length up to 2.000 and evaluated it on the 99 sequences of length 10.000. Then we incorporated one sequence of length 10.000 into the training dataset and evaluated it on the remaining 99 sequences. The predictions for 'job_start' parameter prediction are presented in Figure 11.12. The LSTM and GRU models showed no improvement in predictions. The Transformer model showed an overestimation, which is evident from Figure 11.12f.

11.7.2. Discussion

We can use the models, trained on shorter sequences, to predict the simulation results for longer sequences, as was demonstrated in Chapter 9. However, the accuracy of such predictions depends on platform complexity. The integration of one longer sequence leads to improvement in the model's prediction but does not guarantee accurate predictions.

11.8. Training to Predict One Type of Sequences

In previous experiments, we trained and evaluated the models on datasets with sequences of different lengths. In this evaluation, we aim to assess the accuracy of predictions for extensive sequences comprising 10.000 jobs each, made by models that were trained exclusively on sequences of identical lengths.

The same test dataset was used for all models, and training datasets of each size contained the same sequences for all model types. The evaluation dataset contains 50 sequences with 10.000 jobs each, with the simulation data from Chapter 9. We evaluate the models trained on 6 datasets, containing 1, 10, 20, 30, 40, and 50 sequences with 10.000 jobs each.

11.8.1. Results

The next figures present the predictions for the 'job_start' parameter. Figures 11.13 and 11.14 present the predictions made by the GRU model. Figures 11.15 and 11.16 present the predictions made by the LSTM model. Figures 11.17 and 11.18 present the predictions made by the Transformer model. The predictions 'job_start' and 'job_end' parameters exhibit similar accuracy.

The ’compute_time’ parameter is predicted accurately by the models, trained on one sequence. The predictions for ’input_files_transfer_time’ and ’output_files_transfer_time’ parameters, generated by models trained on 10 sequences, did not show further improvement with training on larger datasets.

11.8.2. Discussion

The GRU model shows the best result among all models, and the model trained on 20 sequences made good predictions and improved the predictions on larger datasets. The LSTM model made accurate predictions trained on 30 sequences, but tended to overestimate and underestimate some jobs, which can be observed in Figures 11.16c and 11.16e. The Transformer model provided accurate predictions, yet they were the least precise compared to those made by GRU and LSTM models. All three models made some inaccurate predictions for the first jobs in the sequence, which can be observed in all accuracy plots in the leftmost part of the plot.

11.9. Comparison with DCSim

In the end, we aim to compare the models with DCSim. We compare the time that is required for DCSim to perform a simulation with time for model training and model inference. The time required for dataset preparation depends directly on time required for DCSim simulations.

Each simulation containing 10.000 jobs with configuration from the third scenario, described in Section 5.4 took between 14 and 25 minutes of CPU time in BwUniCluster, with an average duration of 14,5 minutes. The simulation duration depends heavily on the platform’s complexity.

In Section 11.8, we showed that the GRU model is capable of predicting parameters for sequences of uniform length on a particular platform configuration with a training set of just 20 sequences. In contrast, while the LSTM model required slightly more sequences for training, it also produced very accurate predictions. In Section 11.2 we demonstrated, that all three model types are capable of predicting parameters for sequences of different lengths, trained on a dataset which contained the 50 sequences of each length.

The model training requires less than 2 minutes for this platform. As discussed in Section 11.3, that one training epoch takes less than one second even on the largest dataset. Further in Section 11.4 we found, that all models require less than 100 epochs for training, and the GRU models reduce the loss systematically faster.

In all scenarios, the models demonstrated the ability to predict the job simulation results and match the distribution for the sequences of the known lengths. The results of the second and third scenarios, described in Chapters 8 and 9 demonstrate, that the predictions for longer sequences can include systematical errors for some parameters. The findings of the fourth Scenario, described in Chapter 10 demonstrate, that such models are not suitable for platform generalization with the current strategy.

All models take under 1 second for inference of the one sequence with 10.000 jobs including the data preparation, which was demonstrated in Section 11.5. Once the model is trained, it can generate predictions almost instantaneously.

While models excel in specific configurations and can greatly reduce the time required for predictions compared to a simulation, they are not a direct replacement for simulators and their accuracy is inherently limited to the quality and range of the training data provided. The models developed in this work cannot directly overcome the platform scalability issues of the simulator, but they offer a practical trade-off by being trainable on simulated or historical data, balancing prediction accuracy with enhanced processing speed.

11.10. Limitations

The models are trained and optimized to specific platform and workload configurations tested during the study. This specificity limits their applicability to different configurations not represented in the training data.

The evaluation of the models was constrained to a small number of platform configurations. Real-world distributed systems could exhibit greater complexity and variability, which may challenge the scalability and effectiveness of the models.

The inherent complexity of the employed RNN and Transformer architectures limits the transparency of the predictive processes. This poses challenges to their interpretability and acceptance in practical applications.

The models are primarily designed for numerical predictions. Integration of categorical variables alongside numerical ones presents additional challenges and requires further research and methodological adjustments to ensure accurate modeling. This limitation is particularly relevant when considering the diverse nature of data in distributed computing environments, where categorical variables often play an important role.

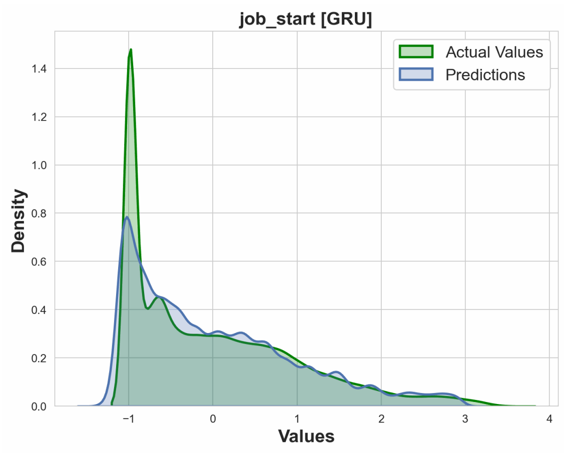
11.11. Threats to Validity

The choice and tuning of hyperparameters might not be optimal for all scenarios tested, potentially leading to overfitting or underfitting. Ensuring the internal validity of our conclusions requires cross-validation and sensitivity analysis to demonstrate that results are not dependent on specific model configurations.

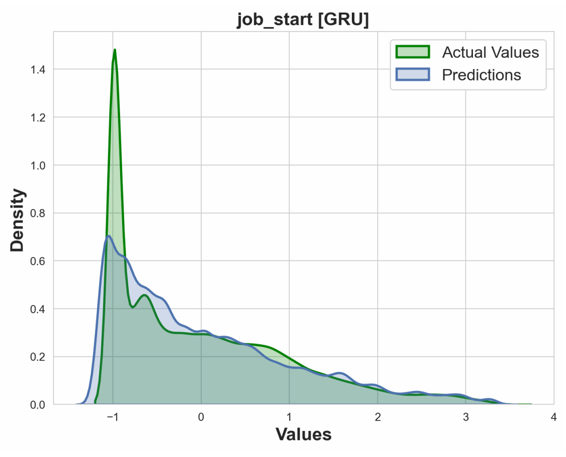
MSE and R-squared provide a single summary statistic that represents the average performance of a model across all predictions. This aggregation can mask variations in prediction accuracy within different segments of long sequences or across different types of job parameters. For example, the model might perform well overall but poorly on certain segments that are critical for the system's performance.

Since the training data is derived from DCSim simulations, any errors in DCSim simulations will also affect the predictions. We used the DCSim simulations as the ground truth. Any inaccuracies in DCSim simulations would lead to biased or incorrect training data, compromising the reliability of the model's predictions.

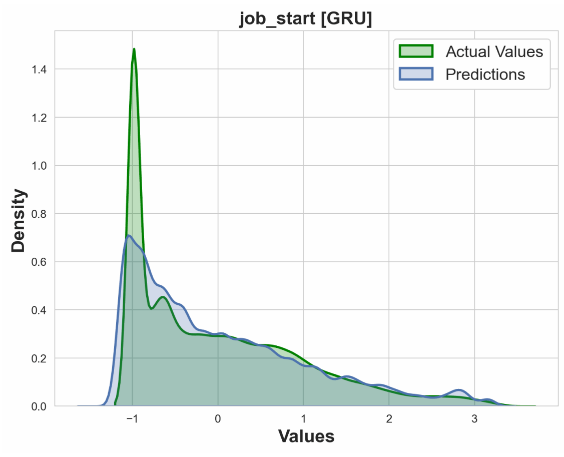
Another threat to validity arises from the limited variety of platforms used in our evaluation, restricted to only three configurations. This limitation raises concerns about the model's ability to generalize for new workloads and accurately predict outcomes for more complex or substantially different platform configurations, which were not represented in the training data. Additionally, the training data consisted of simulated jobs generated from at most five predefined workload classes. This limited variety in workload classes may pose a threat to validity, as the models might not perform as effectively when exposed to job types or classes beyond those included in the training set. The introduction of new workload classes could disrupt the model's prediction accuracy, challenging their robustness and generalizability in more complex real-world environments.



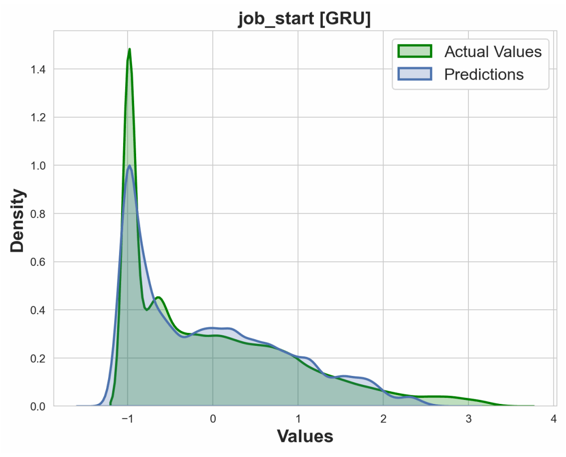
(a) Trained on one sequence of each length



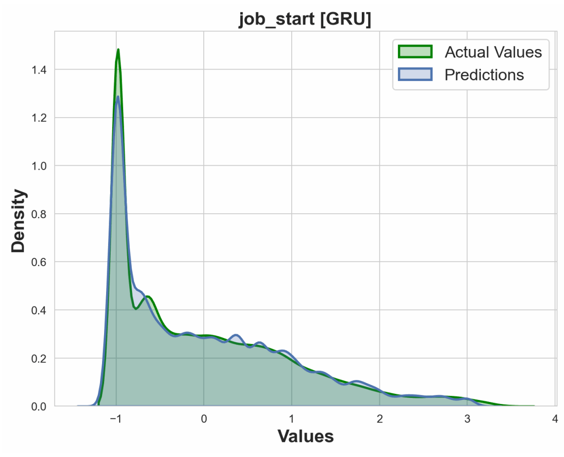
(b) Trained on 10 sequences of each length



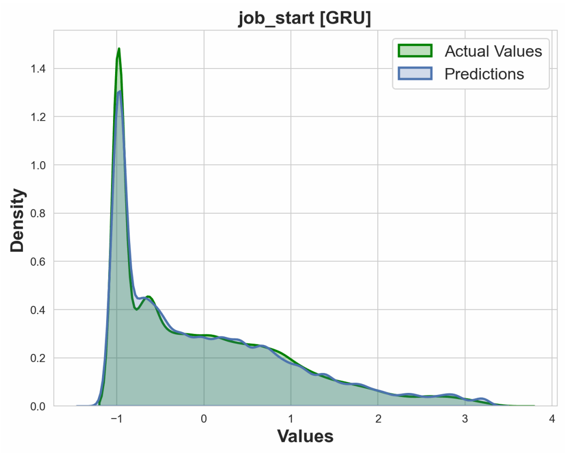
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

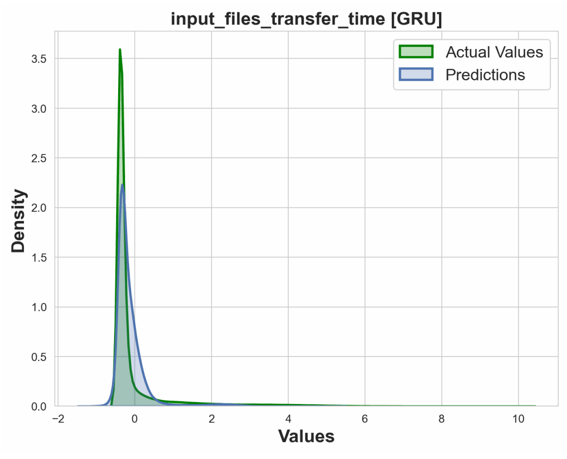


(e) Trained on 40 sequences of each length

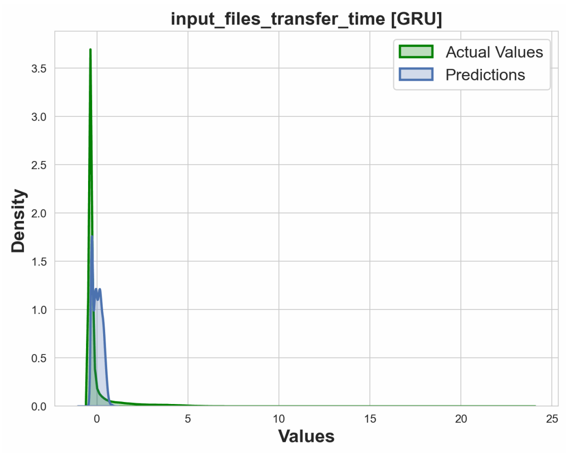


(f) Trained on 50 sequences of each length

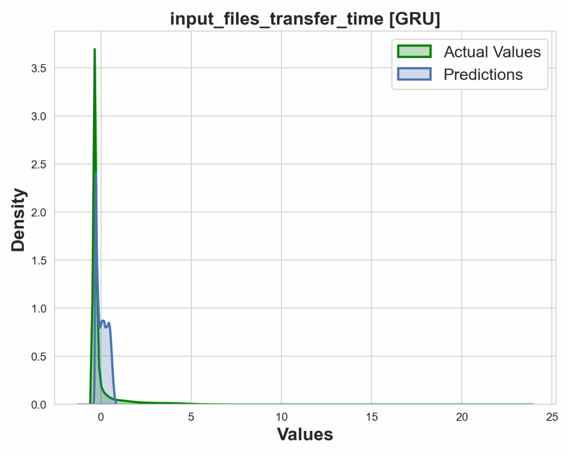
Figure 11.4.: GRU Training Efficiency for 'job_start' Prediction



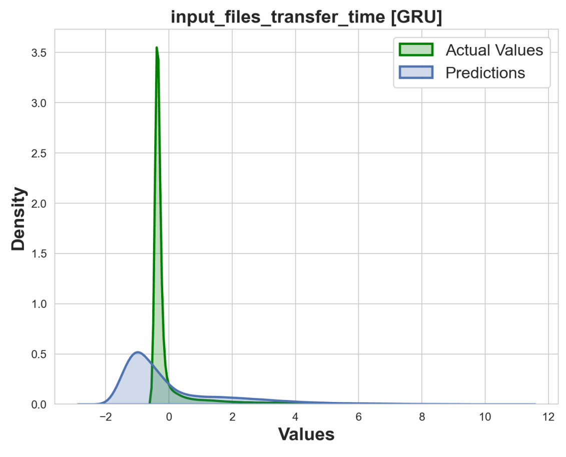
(a) Trained on one sequence of each length



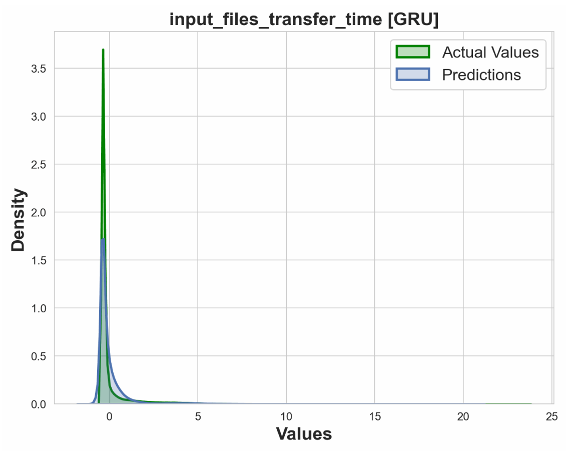
(b) Trained on 10 sequences of each length



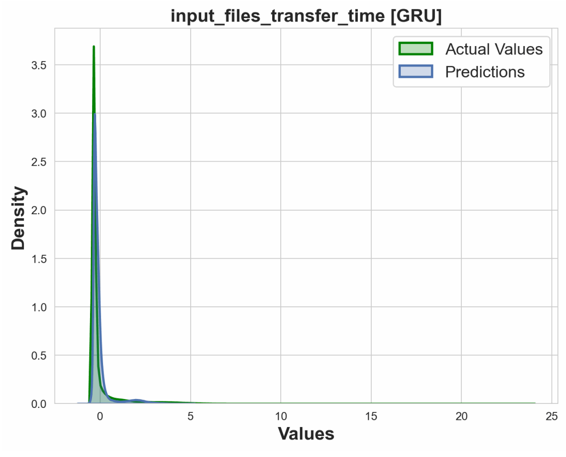
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

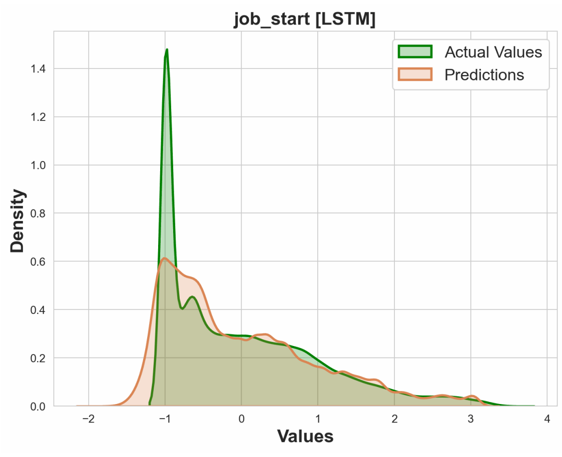


(e) Trained on 40 sequences of each length

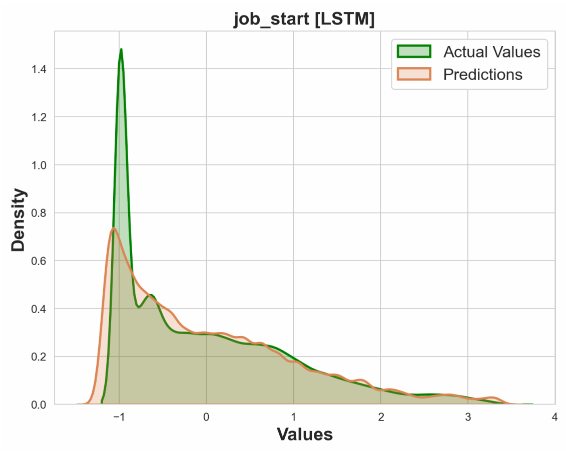


(f) Trained on 50 sequences of each length

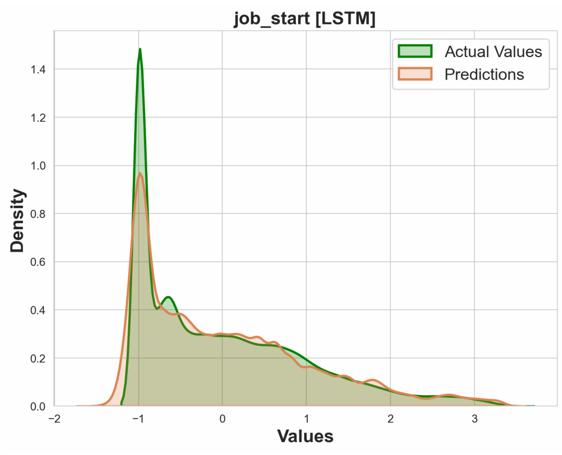
Figure 11.5.: GRU Training Efficiency for 'input_files_transfer_time' Prediction



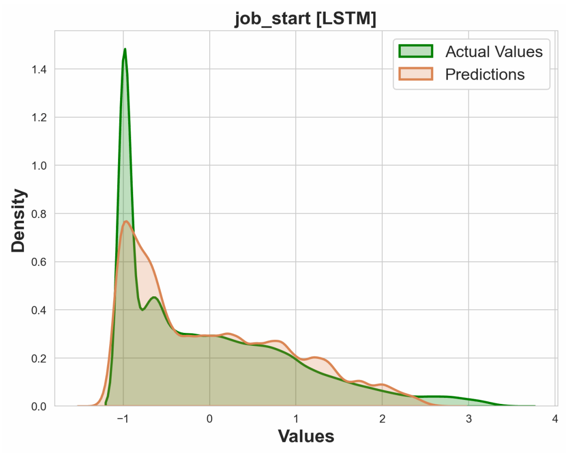
(a) Trained on one sequence of each length



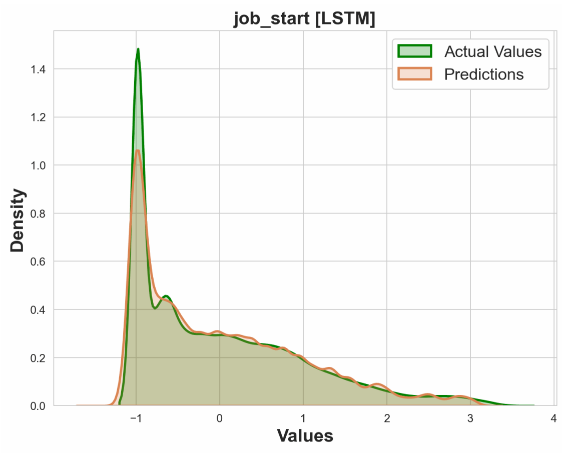
(b) Trained on 10 sequences of each length



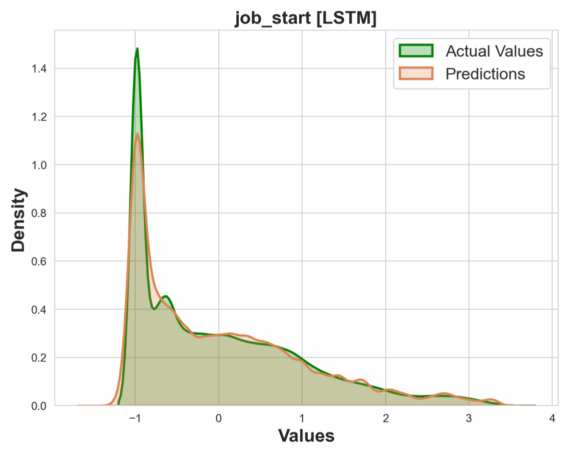
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

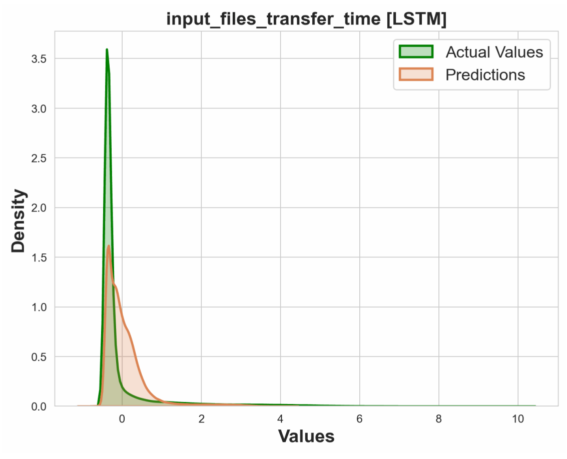


(e) Trained on 40 sequences of each length

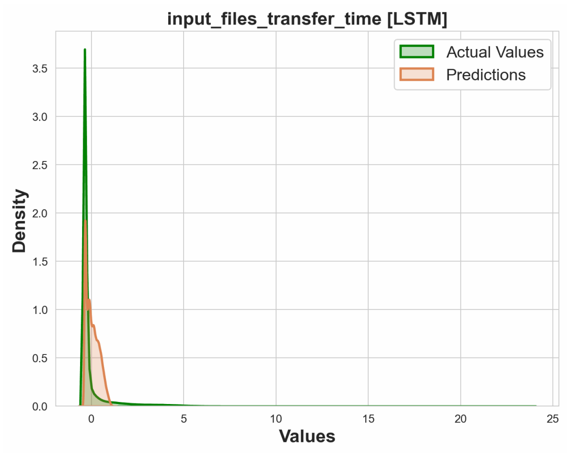


(f) Trained on 50 sequences of each length

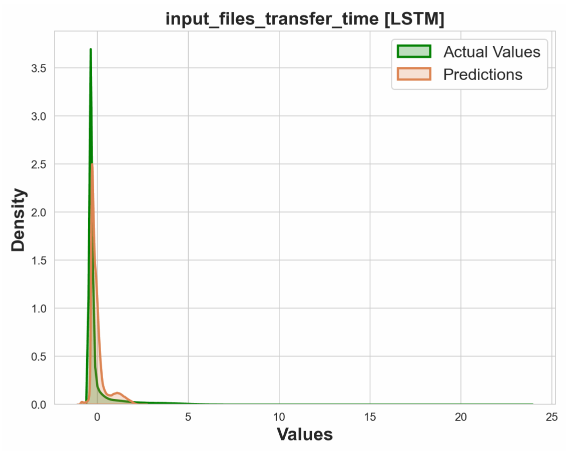
Figure 11.6.: LSTM Training Efficiency for 'job_start' Prediction



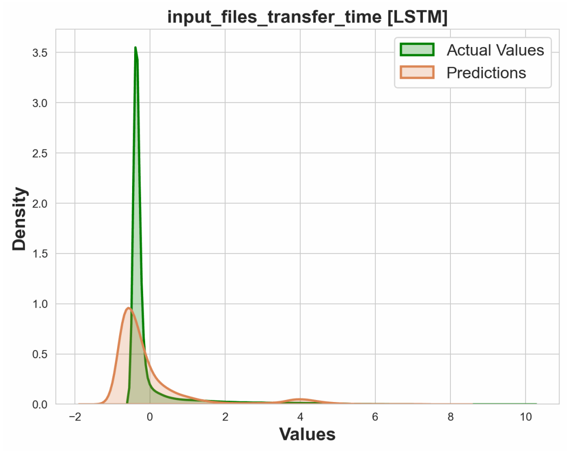
(a) Trained on one sequence of each length



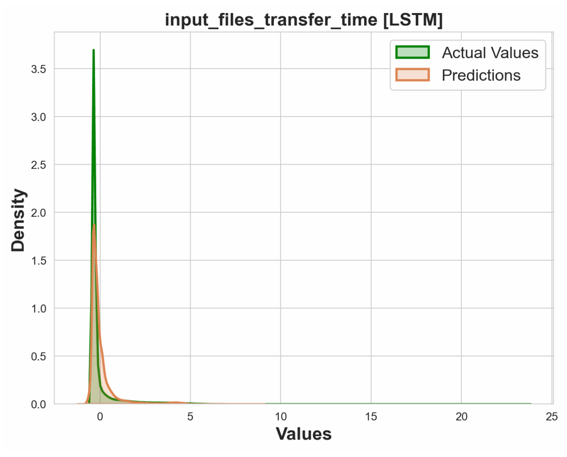
(b) Trained on 10 sequences of each length



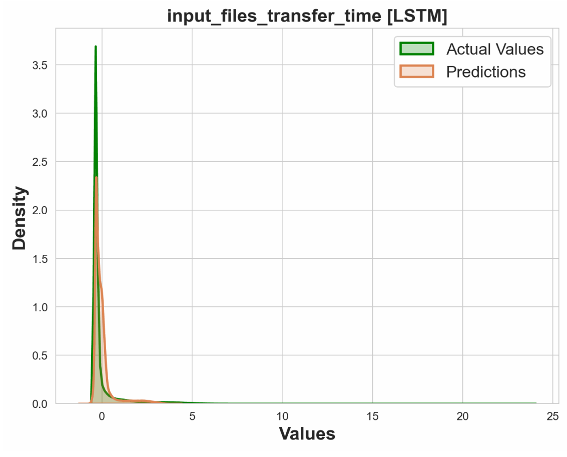
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

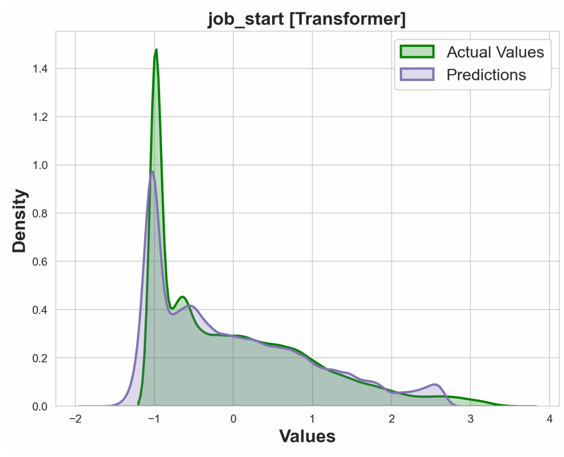


(e) Trained on 40 sequences of each length

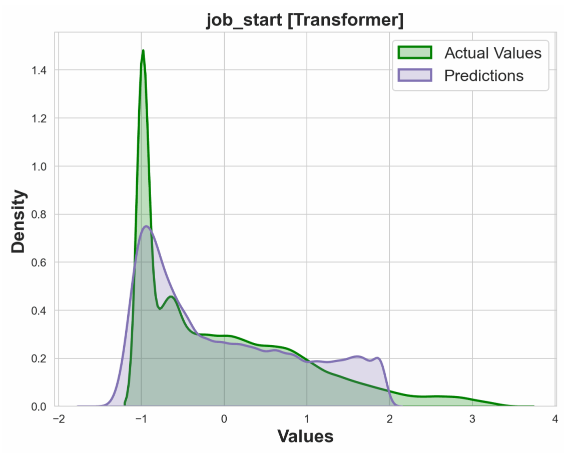


(f) Trained on 50 sequences of each length

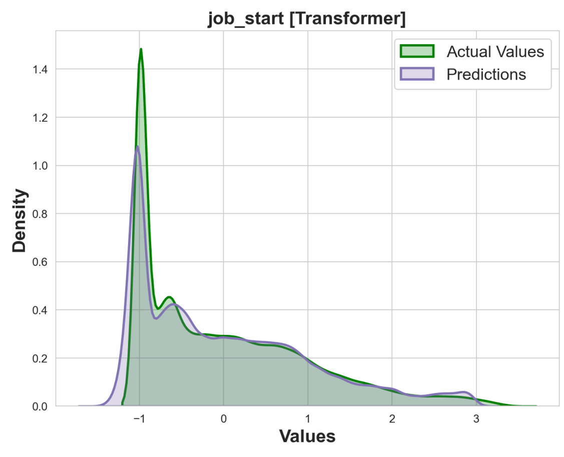
Figure 11.7.: LSTM Training Efficiency for 'input_files_transfer_time' Prediction



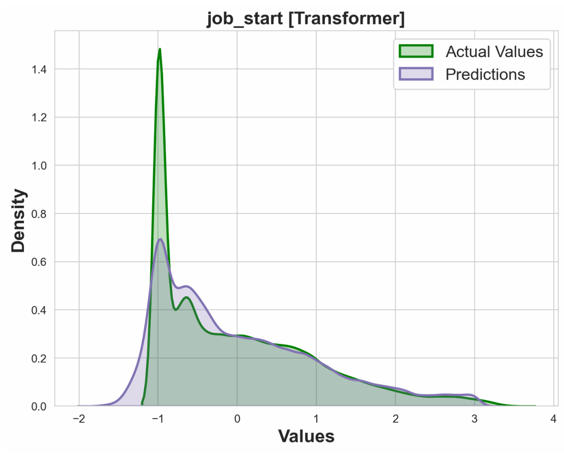
(a) Trained on one sequence of each length



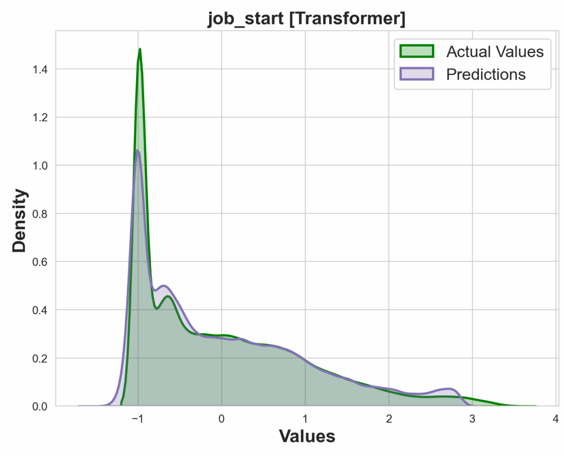
(b) Trained on 10 sequences of each length



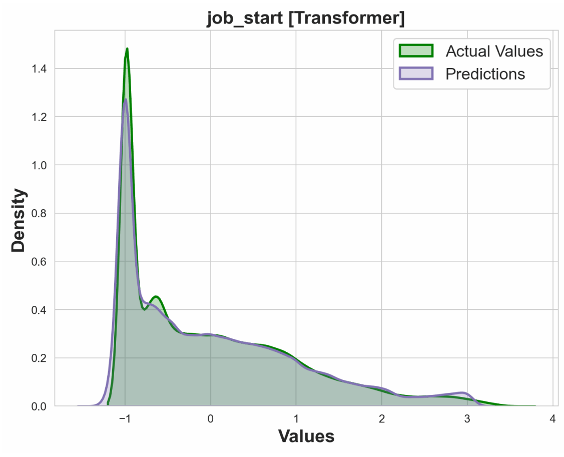
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

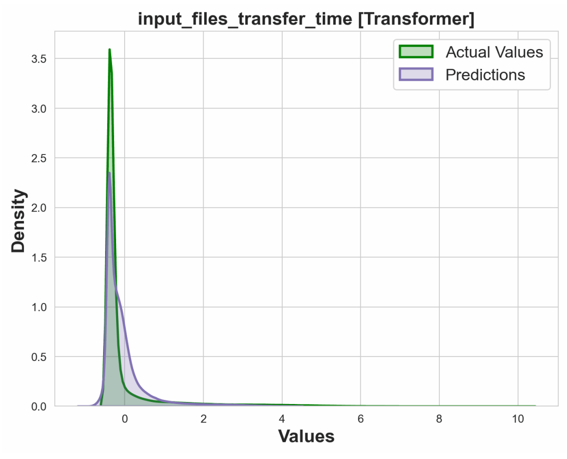


(e) Trained on 40 sequences of each length

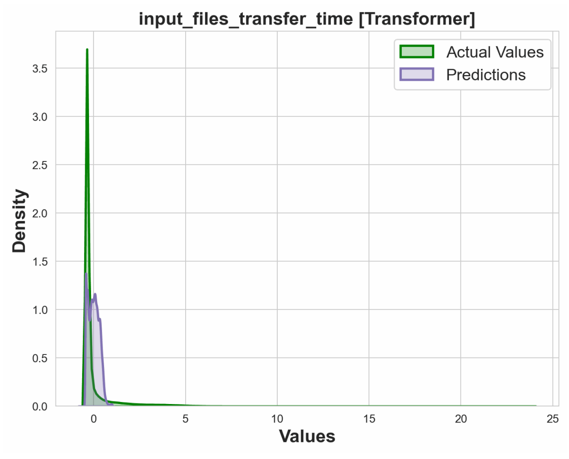


(f) Trained on 50 sequences of each length

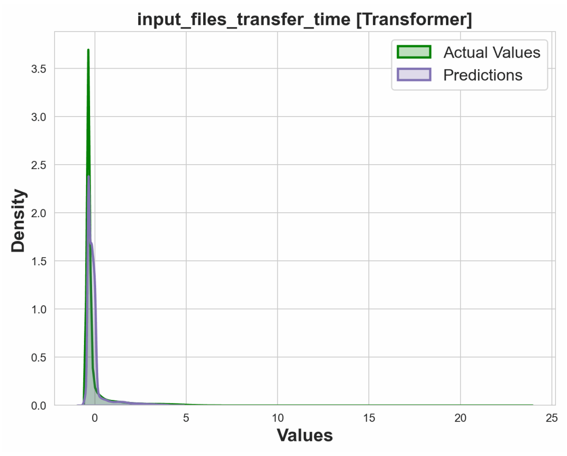
Figure 11.8.: Transformer Training Efficiency for 'job_start' Prediction



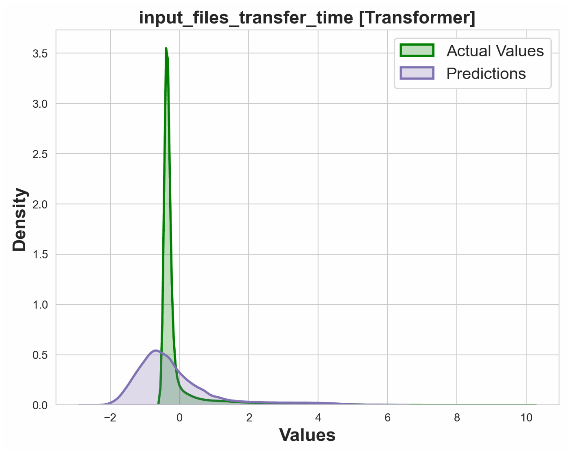
(a) Trained on one sequence of each length



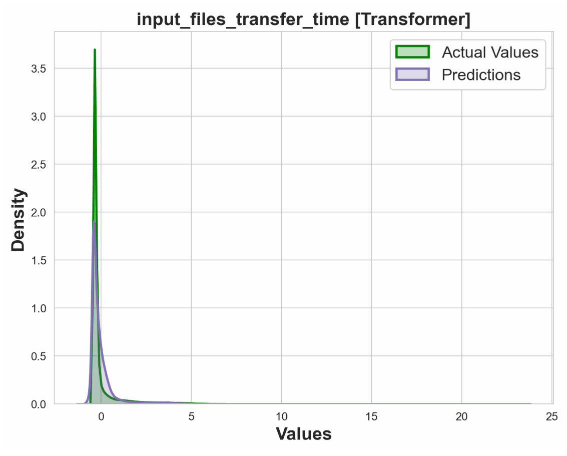
(b) Trained on 10 sequences of each length



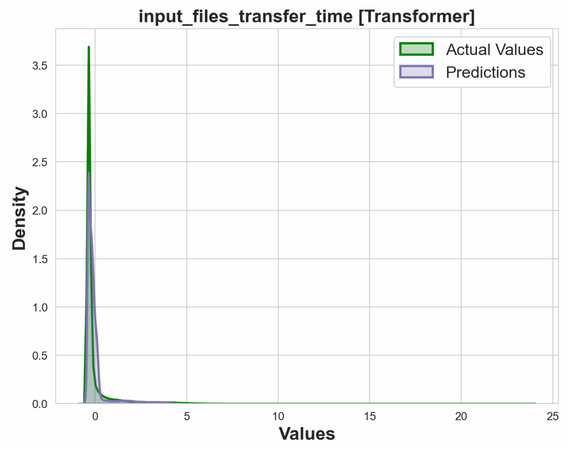
(c) Trained on 20 sequences of each length



(d) Trained on 30 sequences of each length

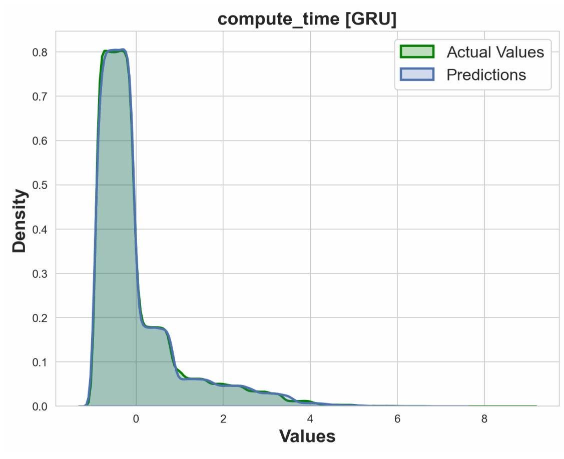


(e) Trained on 40 sequences of each length

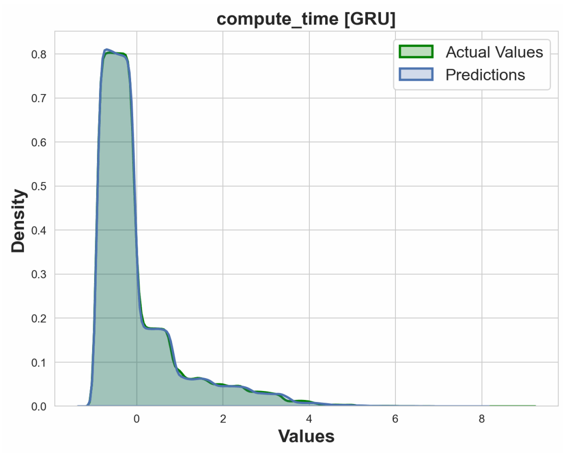


(f) Trained on 50 sequences of each length

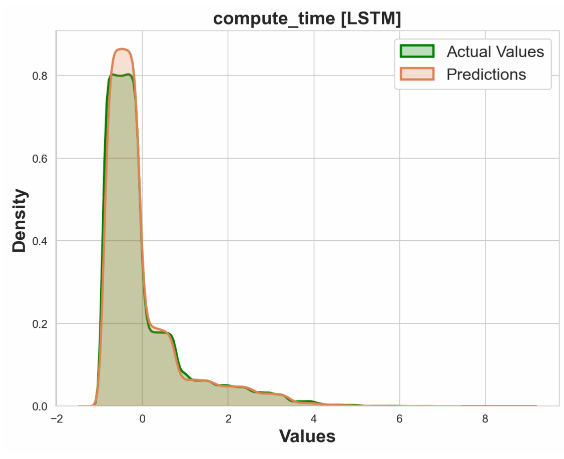
Figure 11.9.: Transformer Training Efficiency for 'input_files_transfer_time' Prediction



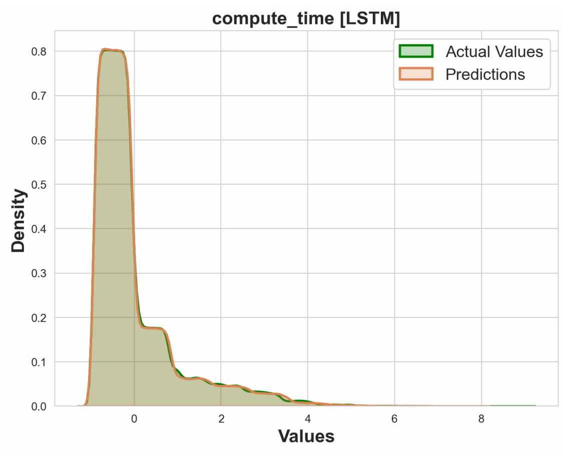
(a) GRU: one sequence of each length



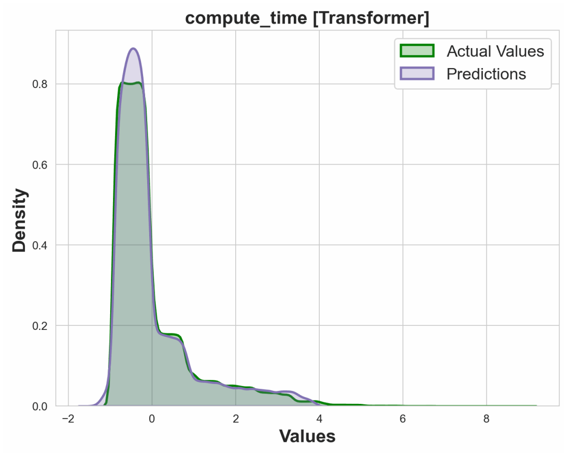
(b) GRU: 500 sequences of each length



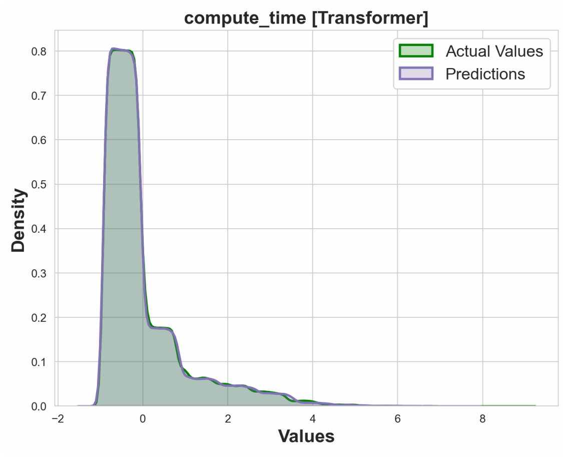
(c) LSTM: one sequence of each length



(d) LSTM: 500 sequences of each length

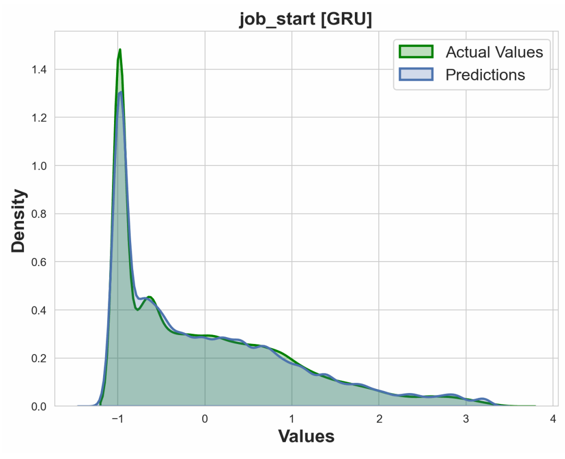


(e) Transformer: one sequence of each length

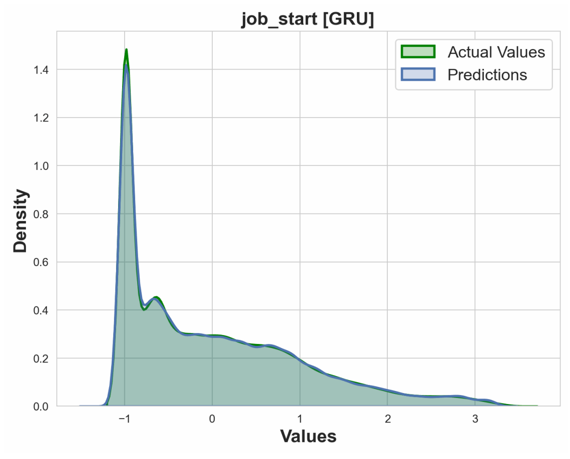


(f) Transformer: 500 sequences of each length

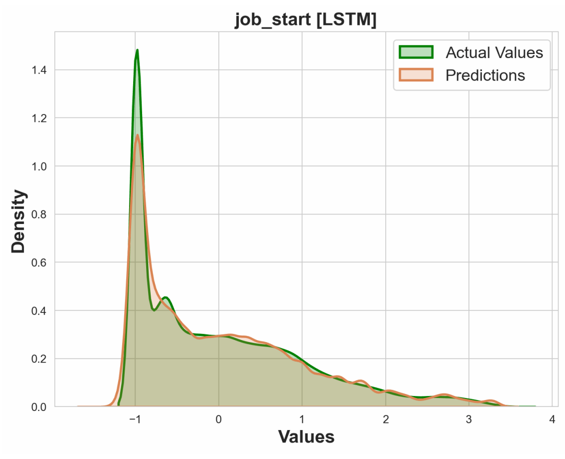
Figure 11.10.: Training Efficiency for 'compute_time' Prediction



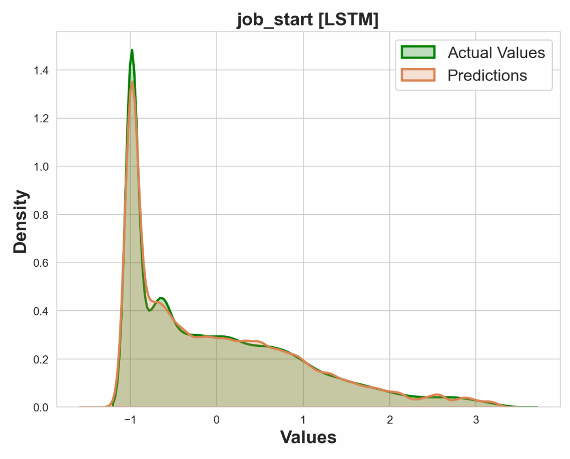
(a) GRU Trained on 50 sequence of each length



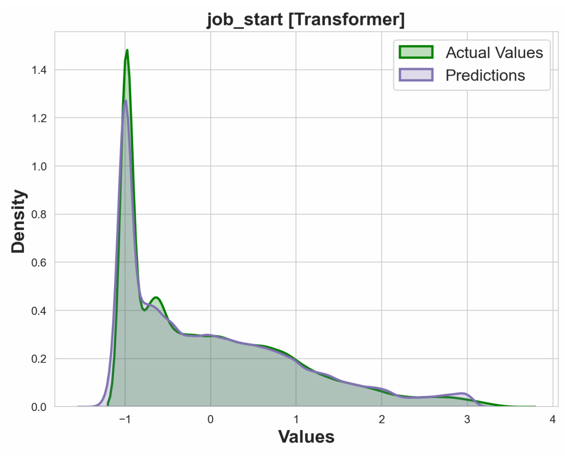
(b) GRU trained on 500 sequences of each length



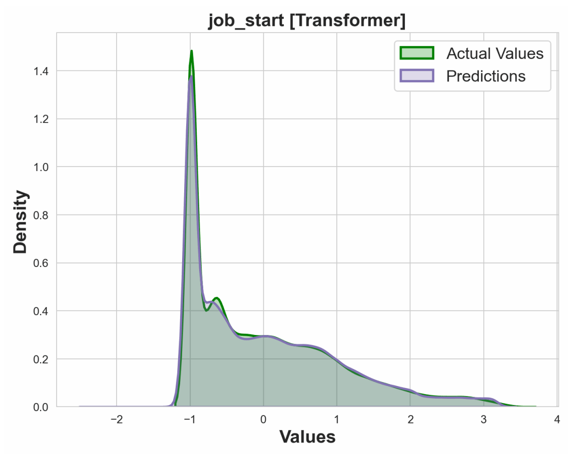
(c) LSTM trained on 50 sequences of each length



(d) LSTM trained on 500 sequences of each length

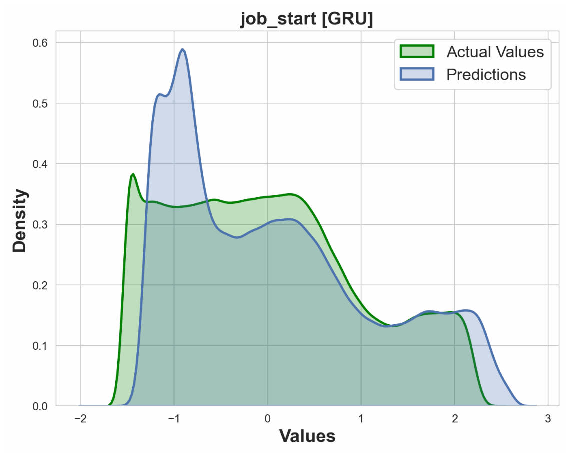


(e) Transformer trained on 50 sequences of each length

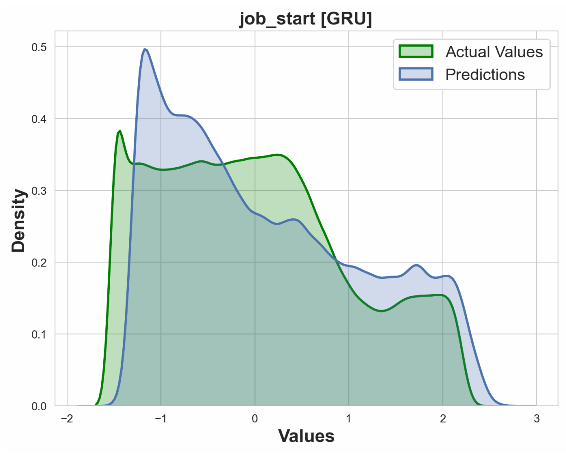


(f) Transformer trained on 500 sequences of each length

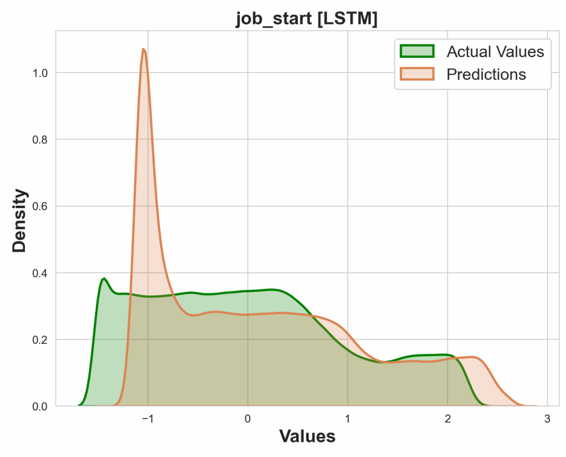
Figure 11.11.: Models trained on larger datasets



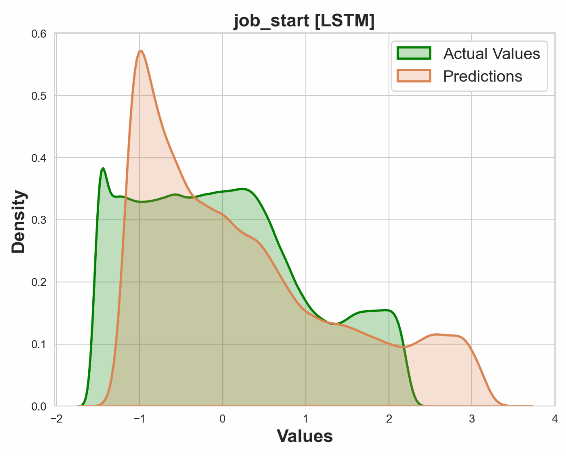
(a) GRU default dataset



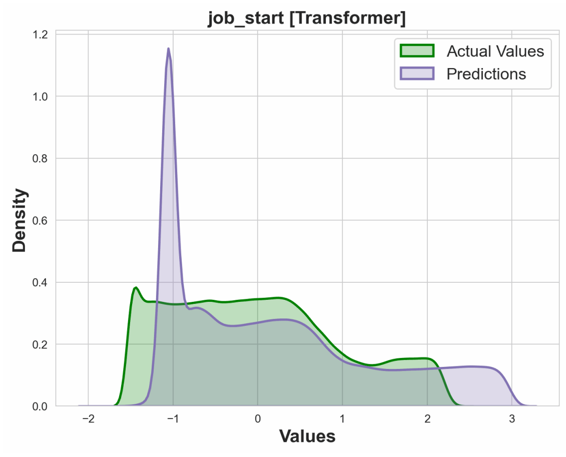
(b) GRU with additional long sequence



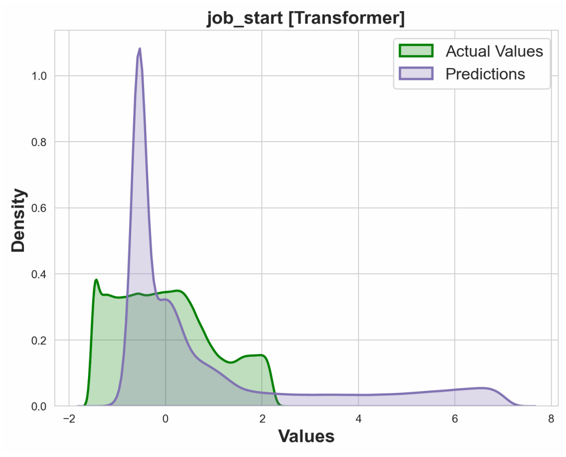
(c) LSTM default dataset



(d) LSTM with additional long sequence

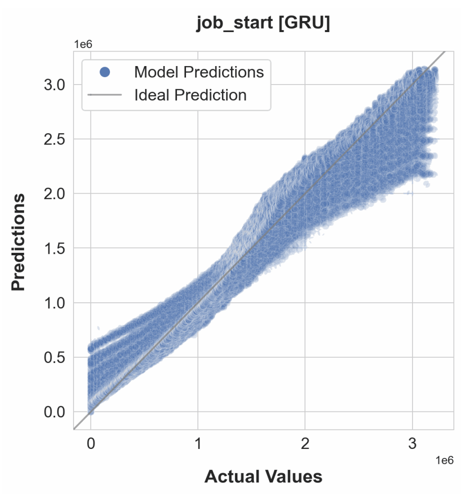


(e) Transformer default dataset

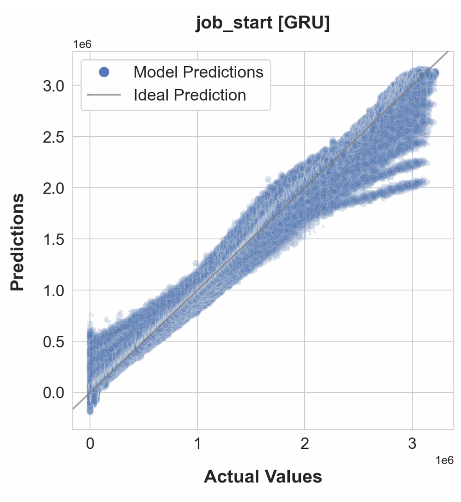


(f) Transformer with additional long sequence

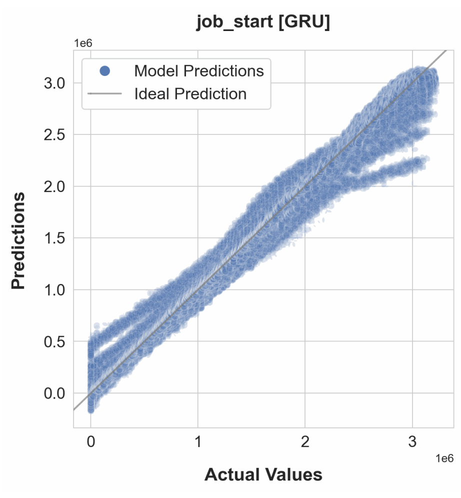
Figure 11.12.: Models trained on the data with additional long sequence



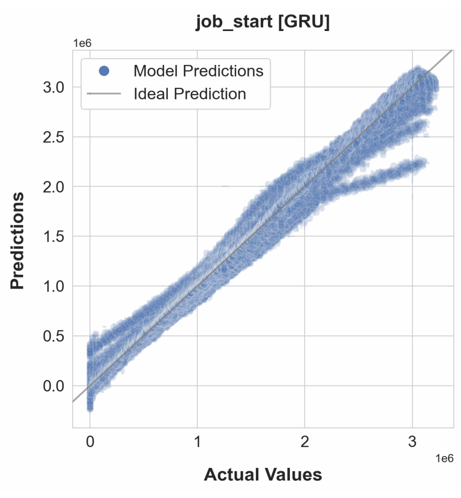
(a) Trained on one sequence



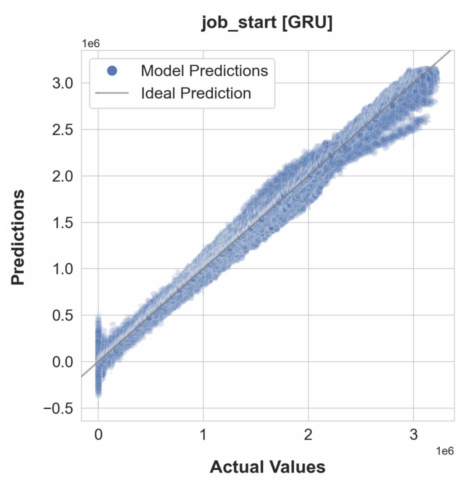
(b) Trained on 10 sequences



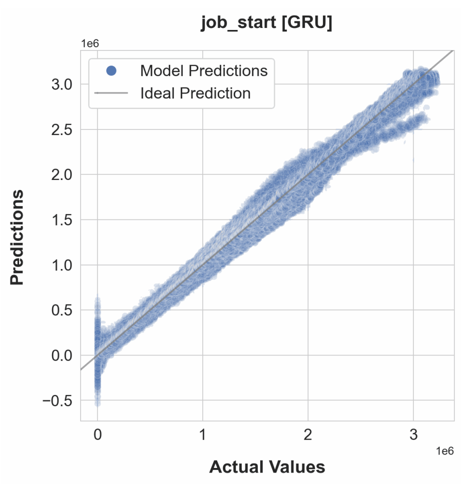
(c) Trained on 20 sequences



(d) Trained on 30 sequences

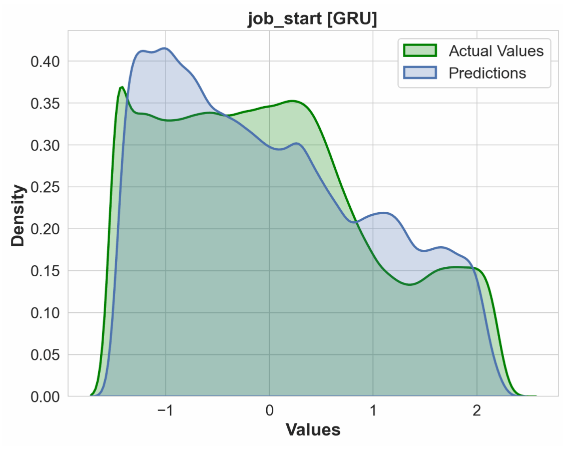


(e) Trained on 40 sequences

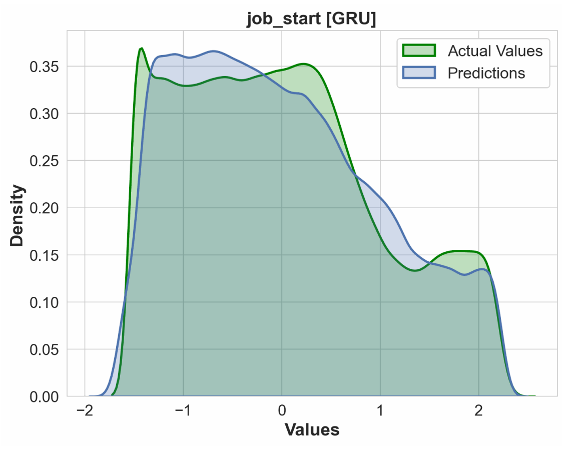


(f) Trained on 50 sequences

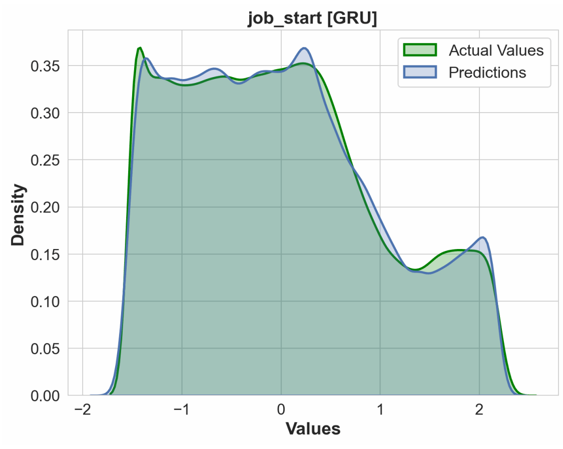
Figure 11.13.: GRU Training Efficiency for 'job_start' Prediction



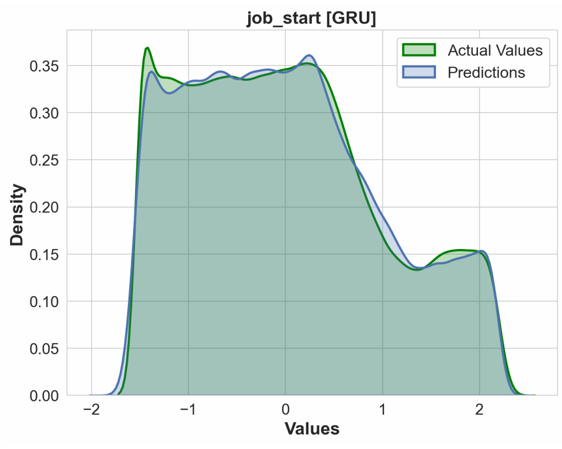
(a) Trained on one sequence



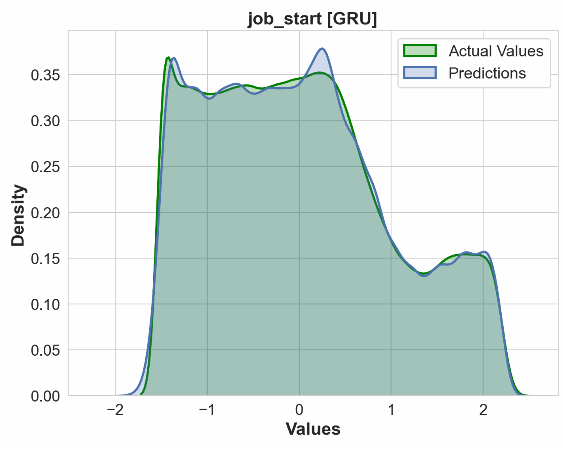
(b) Trained on 10 sequences



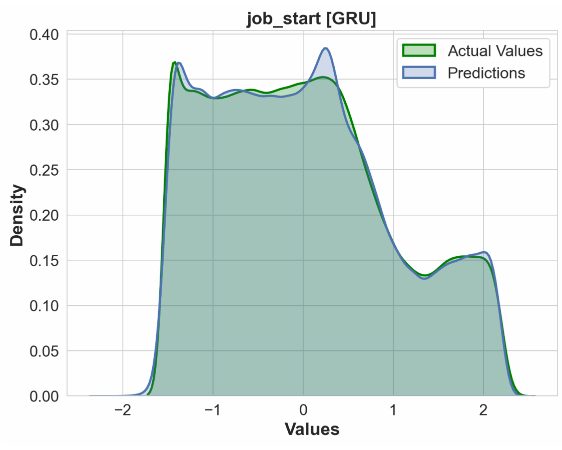
(c) Trained on 20 sequences



(d) Trained on 30 sequences

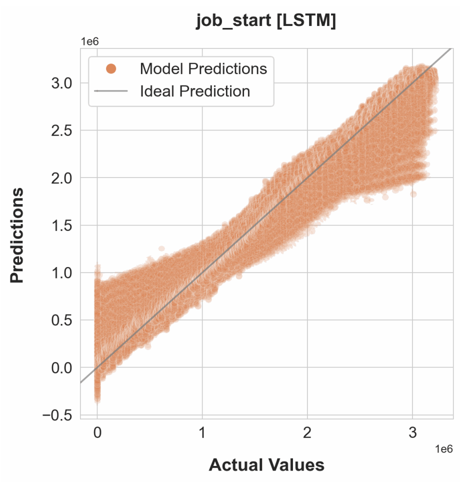


(e) Trained on 40 sequences

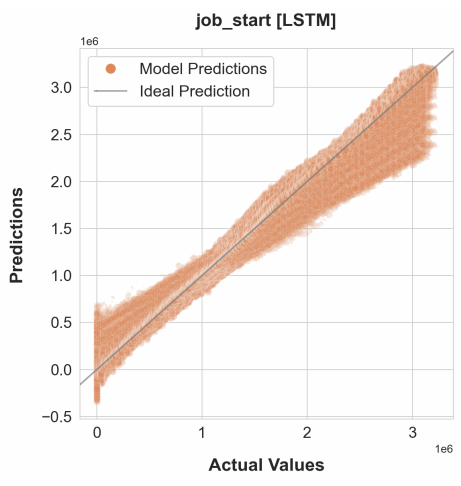


(f) Trained on 50 sequences

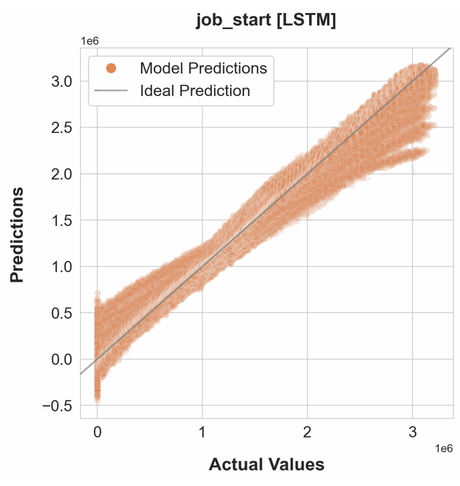
Figure 11.14.: GRU Training Efficiency for 'job_start' Prediction (KDE)



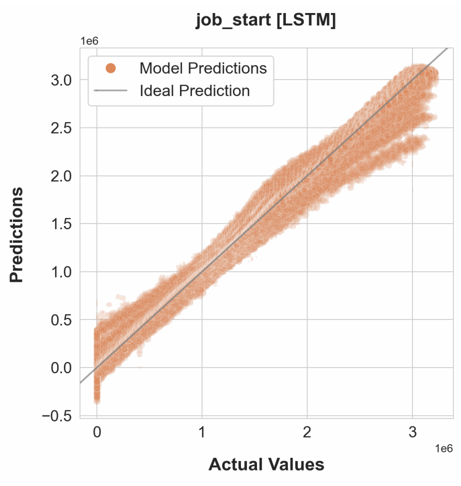
(a) Trained on one sequence



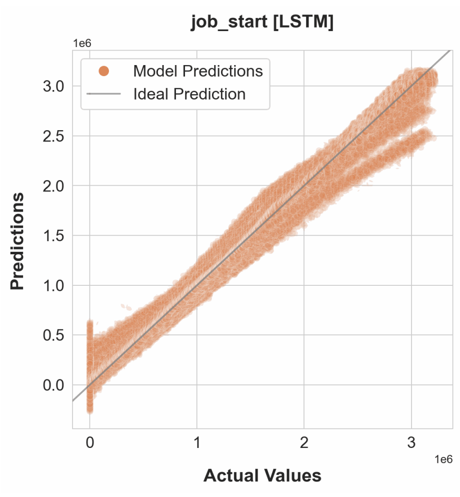
(b) Trained on 10 sequences



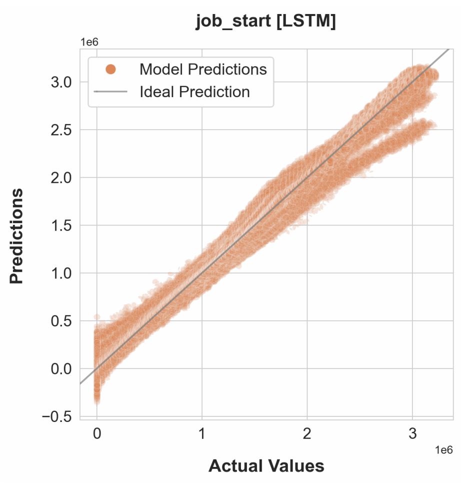
(c) Trained on 20 sequences



(d) Trained on 30 sequences

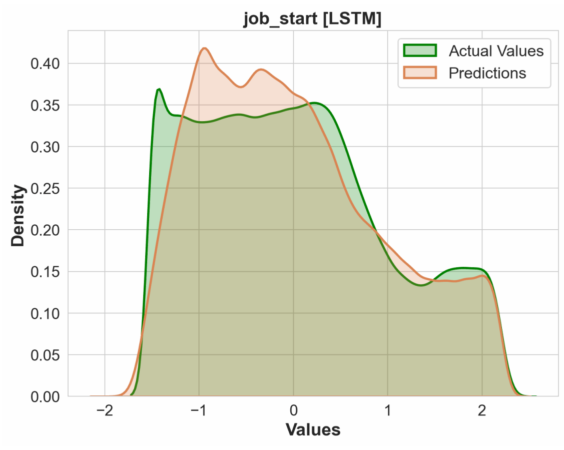


(e) Trained on 40 sequences

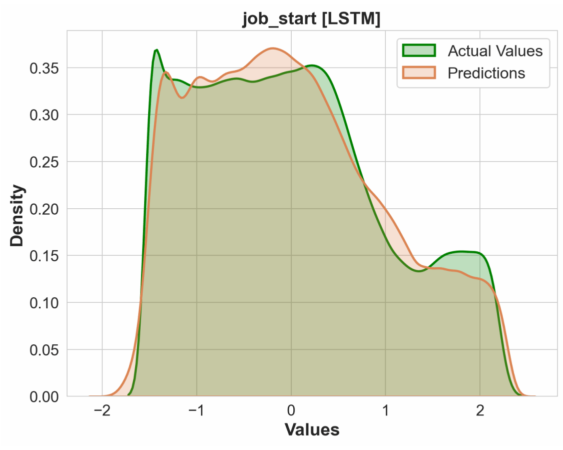


(f) Trained on 50 sequences

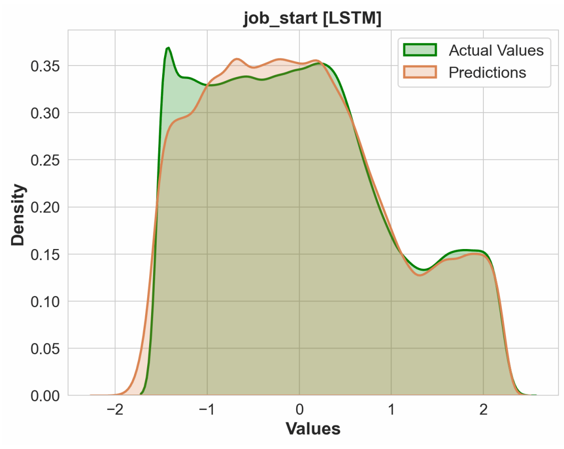
Figure 11.15.: LSTM Training Efficiency for 'job_start' Prediction



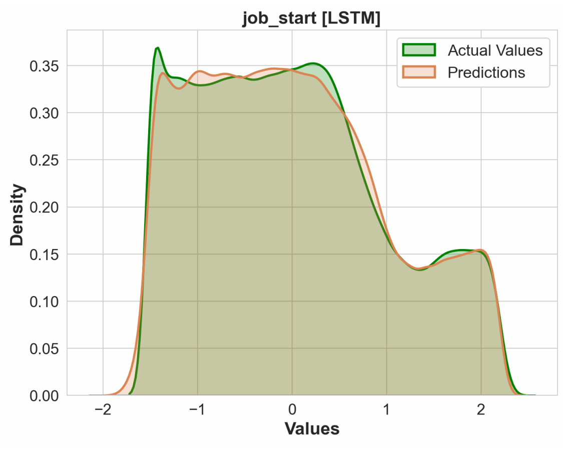
(a) Trained on one sequence



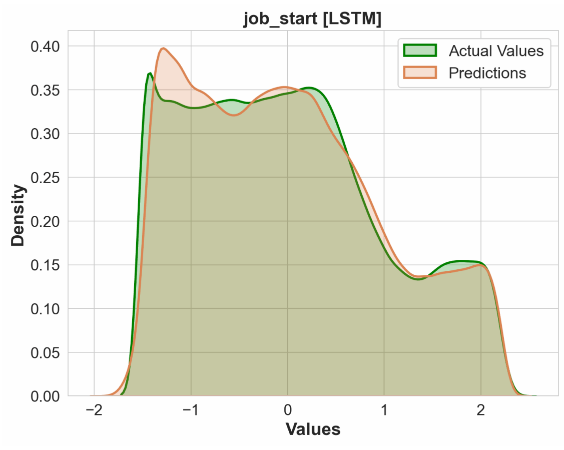
(b) Trained on 10 sequences



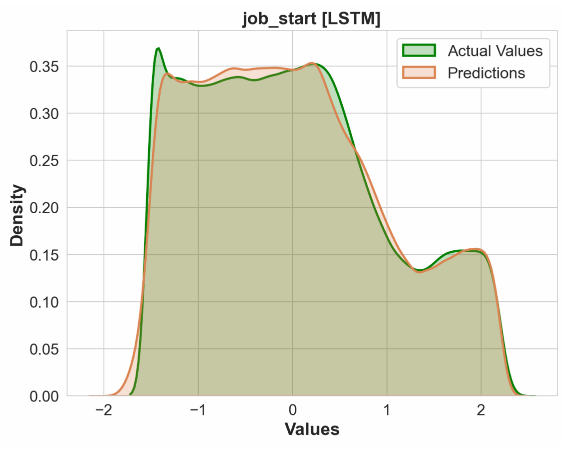
(c) Trained on 20 sequences



(d) Trained on 30 sequences

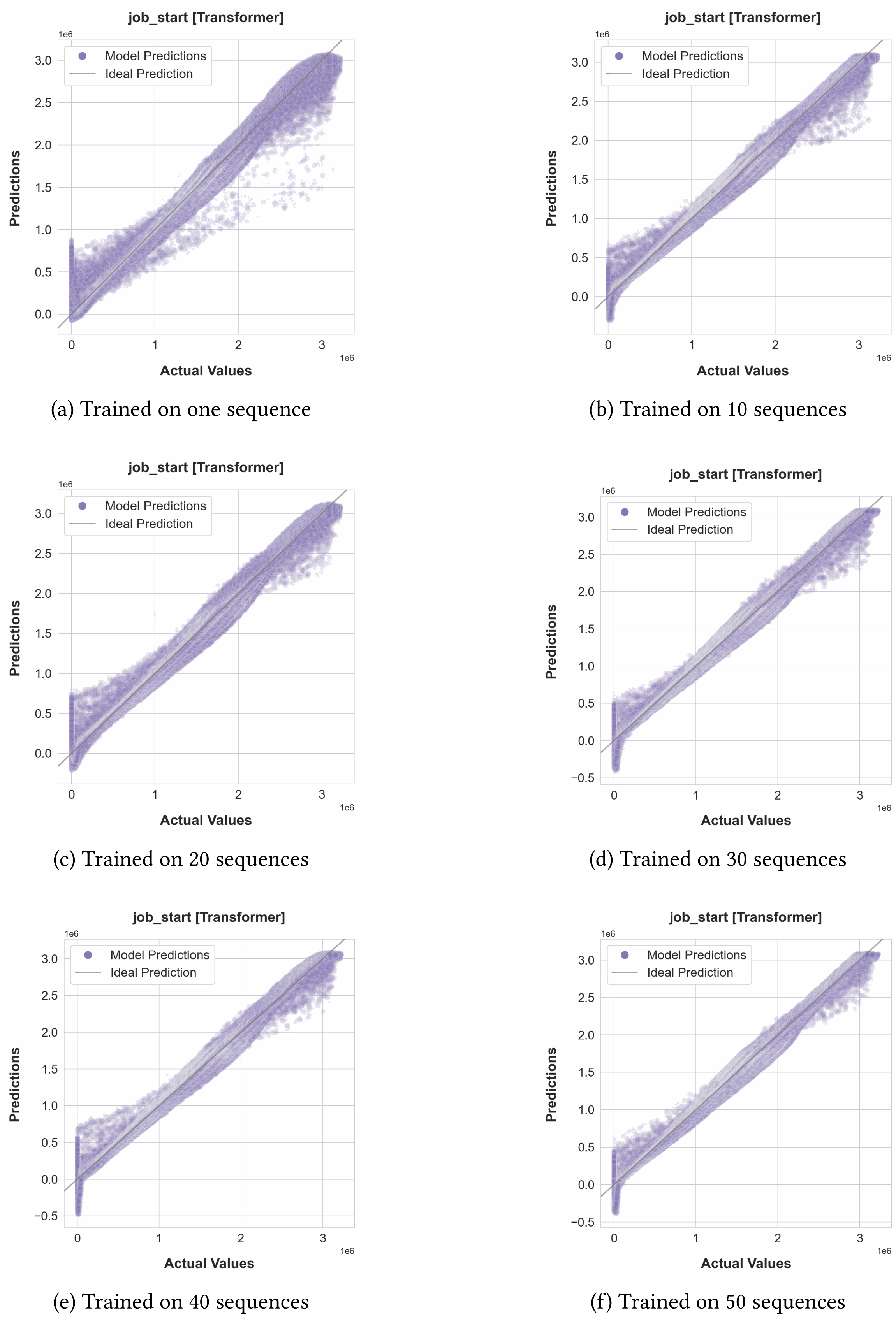


(e) Trained on 40 sequences



(f) Trained on 50 sequences

Figure 11.16.: LSTM Training Efficiency for 'job_start' Prediction (KDE)



(a) Trained on one sequence

(b) Trained on 10 sequences

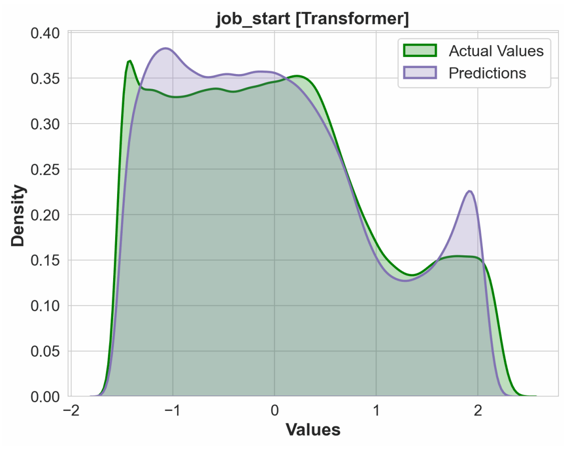
(c) Trained on 20 sequences

(d) Trained on 30 sequences

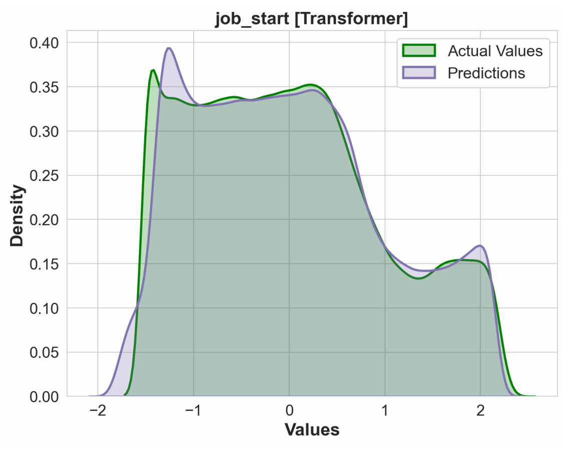
(e) Trained on 40 sequences

(f) Trained on 50 sequences

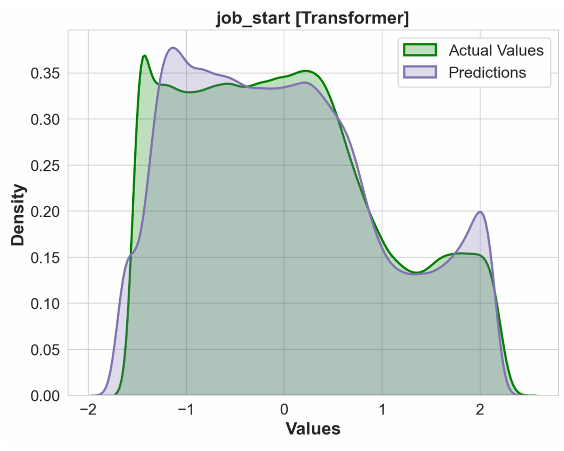
Figure 11.17.: Transformer Training Efficiency for 'job_start' Prediction



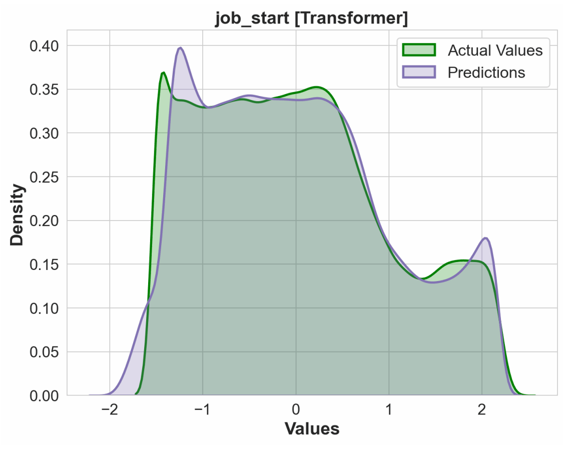
(a) Trained on one sequence



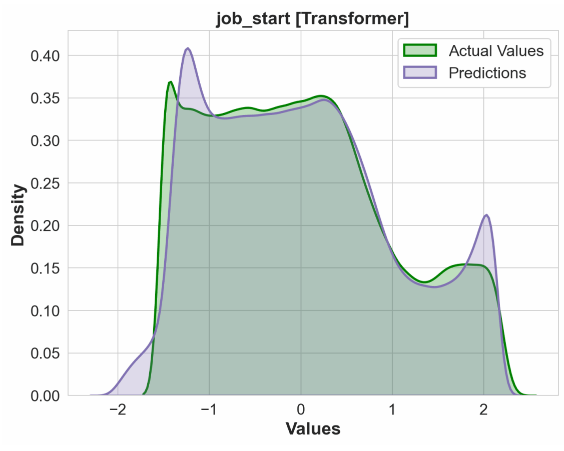
(b) Trained on 10 sequences



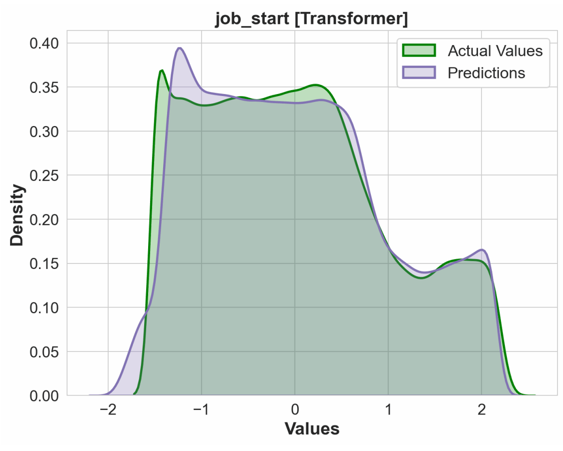
(c) Trained on 20 sequences



(d) Trained on 30 sequences



(e) Trained on 40 sequences



(f) Trained on 50 sequences

Figure 11.18.: Transformer Training Efficiency for 'job_start' Prediction

12. Conclusion

In this thesis, we explored the application of RNN and Transformer networks to predict job execution metadata in distributed computing systems, developing a flexible methodology for numerical sequence-to-sequence predictions. The models require training data, which can be obtained either from the simulator or from actual job processing. We employed the DCSim to perform the simulations of WLCG workloads on GridKa infrastructure. Since DCSim simulation results share a similar format for job inputs and outputs as WLCG logs, the concepts we propose in this thesis can be adapted to real-world data.

In our research, we created datasets with simulation data for three distinct scenarios to train our models. These scenarios represent two real data centers and workloads with increasing complexity. To facilitate this process, we developed an automated infrastructure for data generation and preparation.

We created models with three distinct architectures — BiGRU, BiLSTM, and Encoder-Only Transformer — and evaluated their performance on two tasks: interpolation and extrapolation. Interpolation tasks involved evaluating models on unseen simulations of lengths (number of jobs in simulation) observed in the training set, while extrapolation tasks tested the model's ability to predict outcomes of longer simulations. In these scenarios, distinct models were trained for specific platforms and demonstrated very good results in interpolation tasks and good results in extrapolation tasks.

Additionally, we attempted to incorporate information about simulated platforms (from a small data center to a large grid) into the models to achieve generalizability across different platforms. Access to platform information during model training is important as job input parameters alone do not provide all the essential details required for accurate predictions. We faced challenges with the scaling of numerical platform information and dealing with categorical variables. Training a distinct model for each platform emerged as the most practical approach in our evaluation.

During the evaluation, BiGRU models demonstrated slightly better performance in both training rate and accuracy of predictions among the models evaluated. The BiLSTM and Encoder-Only Transformer models also delivered good results. Once trained with a sufficient amount of data, these models can generate predictions within a few milliseconds, making them useful for integration into scheduling algorithms where rapid cost evaluation is critical. Our findings demonstrate that while these models offer substantial benefits in speed and efficiency, they can not completely replace traditional simulators due to their limitations in handling complex system dynamics.

12.1. Future Work

We used the data generated by the DCSim simulator for model training and evaluation. The next research step would be applying these models to process and predict outcomes based on real-world data, exploring their prediction capabilities and generalization in practical distributed computing environments.

An interesting area for future research lies in exploring the use of Encoder-Decoder Transformers for making predictions on complete sequences without the need for segmenting them into smaller parts. Implementing this approach would necessitate autoregressive generation, where predictions from the last jobs are used as inputs for subsequent jobs.

A. Appendix

A.1. Platform Configurations

Listing A.1: Platform Configuration for Sgbatch

```xml
<?xml version="1.0"?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
<platform version="4.1">
    <config>
        <prop id="network/loopback-bw" value="1000000000000"/>
    </config>

    <zone id="global" routing="Full">
        <zone id="ETP" routing="Floyd">
            <host id="sg01" speed="1117Mf" core="24">
                <prop id="type" value="worker,cache"/>
                <prop id="ram" value="64GiB"/>
                <disk id="ssd_cache1" read_bw="9.6Gbps" write_bw="9.6Gbps">
                    <prop id="size" value="2.5TB"/>
                    <prop id="mount" value="/"/>
                </disk>
            </host>
            <host id="sg02" speed="1117Mf" core="24">
                <prop id="type" value="worker,cache"/>
                <prop id="ram" value="64GiB"/>
                <disk id="ssd_cache1" read_bw="9.6Gbps" write_bw="9.6Gbps">
                    <prop id="size" value="2.5TB"/>
                    <prop id="mount" value="/"/>
                </disk>
            </host>
            <host id="sg03" speed="1258Mf" core="12">
                <prop id="type" value="worker,cache"/>
                <prop id="ram" value="32GiB"/>
                <disk id="ssd_cache1" read_bw="9.6Gbps" write_bw="9.6Gbps">
                    <prop id="size" value="2.5TB"/>
                    <prop id="mount" value="/"/>
                </disk>
            </host>
            <host id="WMSHost" speed="10Gf" core="10">
                <prop id="type" value="scheduler,executor"/>
                <prop id="ram" value="16GB"/>
            </host>

            <router id="etpgateway"/>

            <link id="loopback" bandwidth="5000GBps" latency="0us"/>
            <link id="etp_link0" bandwidth="6Gbps" latency="0us"/>
            <link id="etp_link1" bandwidth="6Gbps" latency="0us"/>
            <link id="etp_link2" bandwidth="6Gbps" latency="0us"/>
            <link id="etp_link3" bandwidth="6Gbps" latency="0us"/>

            <route src="etpgateway" dst="WMSHost">
                <link_ctn id="etp_link0"/>
            </route>
            <route src="etpgateway" dst="sg01">
                <link_ctn id="etp_link1"/>
            </route>
            <route src="etpgateway" dst="sg02">
                <link_ctn id="etp_link2"/>
            </route>
            <route src="etpgateway" dst="sg03">
                <link_ctn id="etp_link3"/>
            </route>
        </zone>

        <zone id="Remote" routing="Full">
            <host id="RemoteStorage" speed="1000Gf" core="10">
                <prop id="type" value="storage"/>
                <disk id="hard_drive" read_bw="40Gbps" write_bw="40Gbps">
                    <prop id="size" value="1PB"/>
                    <prop id="mount" value="/"/>
                </disk>
            </host>

            <link id="etp_to_remote" bandwidth="10Gbps" latency="0us"/>
        </zone>

        <zoneRoute src="ETP" dst="Remote" gw_src="etpgateway" gw_dst="RemoteStorage"
            >
            <link_ctn id="etp_to_remote"/>
        </zoneRoute>
    </zone>
</platform>
```

Listing A.2: Platform Configuration for GridKa and DESY

```
<?xml version="1.0"?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
```

```xml
<platform version="4.1">
    <config>
        <prop id="network/loopback-bw" value="1000000000000"/>
    </config>

    <zone id="global" routing="Floyd">

        <zone id="KIT" routing="Floyd">

            <zone id="GridKA" routing="Floyd">

                <cluster id="Tier1" prefix="Tier1" radical="0-9" suffix="" speed="
                    2555Mf" core="42" bw="1150Mbps" lat="0us">
                    <prop id="type" value="worker"/>
                    <prop id="ram" value="1187.20GiB"/>
                </cluster>

                <zone id="GridKA-service" routing="Floyd">

                    <host id="GridKA_dCache" speed="1000Gf" core="10">
                        <prop id="type" value="storage"/>
                        <disk id="hard_drive" read_bw="920Mbps" write_bw="920Mbps">
                            <prop id="size" value="7PB"/>
                            <prop id="mount" value="/"/>
                        </disk>
                    </host>

                    <host id="WMSHost" speed="10Gf" core="10">
                        <prop id="type" value="scheduler,executor"/>
                        <prop id="ram" value="16GB"/>
                    </host>

                    <router id="GridKAgateway"/>

                    <link id="GridKA_sched" bandwidth="115Mbps" latency="0us"/>

                    <link id="GridKA_Tier1_FATPIPE" bandwidth="1150Mbps" latency="0
                        us" sharing_policy="FATPIPE"/>
                    <link id="GridKA_Tier1" bandwidth="2300Mbps" latency="0us"/>

                    <link id="GridKA_dcachepool_FATPIPE" bandwidth="460Mbps" latency
                        ="0us" sharing_policy="FATPIPE"/>
                    <link id="GridKA_dcachepool" bandwidth="920Mbps" latency="0us"/>

                    <route src="GridKAgateway" dst="WMSHost">
                        <link_ctn id="GridKA_sched"/>
                    </route>

                    <route src="GridKAgateway" dst="GridKA_dCache">
                        <link_ctn id="GridKA_dcachepool_FATPIPE"/>
                        <link_ctn id="GridKA_dcachepool"/>
                    </route>
                </zone>

                <zoneRoute src="GridKA-service" dst="Tier1" gw_src="GridKAgateway" gw_dst="Tier1Tier1_router">
                    <link_ctn id="GridKA_Tier1_FATPIPE"/>
                    <link_ctn id="GridKA_Tier1"/>
                </zoneRoute>

            </zone>


            <zone id="KITcentral" routing="Floyd">

                <router id="KITgateway"/>

                <link id="GridKA_to_KIT" bandwidth="1150Mbps" latency="0us"/>
                <link id="KIT_to_DESY" bandwidth="115Mbps" latency="0us"/>

            </zone>


            <zoneRoute src="GridKA" dst="KITcentral" gw_src="GridKAgateway" gw_dst="KITgateway">
                <link_ctn id="GridKA_to_KIT"/>
            </zoneRoute>

        </zone>


        <zone id="DESY" routing="Floyd">

            <zone id="DESYGrid" routing="Floyd">

                <host id="Tier2" speed="2209Mf" core="200">
                    <prop id="type" value="worker"/>
                    <prop id="ram" value="500GiB"/>
                </host>

                <host id="DESY_dCache" speed="1000Gf" core="10">
                    <prop id="type" value="cache"/>
                    <disk id="hard_drive" read_bw="920Mbps" write_bw="920Mbps">
                        <prop id="size" value="7PB"/>
                        <prop id="mount" value="/"/>
                    </disk>
                </host>

                <router id="DESYGridgateway"/>

                <link id="DESY_Tier2" bandwidth="460Mbps" latency="0us"/>

                <link id="DESY_dCachepool" bandwidth="460Mbps" latency="0us"/>

                <route src="DESYGridgateway" dst="Tier2">
                    <link_ctn id="DESY_Tier2"/>
                </route>

                <route src="DESYGridgateway" dst="DESY_dCache">
                    <link_ctn id="DESY_dCachepool"/>
                </route>

            </zone>

            <zone id="DESYcentral" routing="Floyd">

                <router id="DESYgateway"/>

                <link id="DESYGrid_to_DESY" bandwidth="1150Mbps" latency="0us"/>

            </zone>

            <zoneRoute src="DESYGrid" dst="DESYcentral" gw_src="DESYGridgateway"
                gw_dst="DESYgateway">
                <link_ctn id="DESYGrid_to_DESY"/>
            </zoneRoute>

        </zone>

        <zoneRoute src="KIT" dst="DESY" gw_src="KITgateway" gw_dst="DESYgateway">
            <link_ctn id="KIT_to_DESY"/>
        </zoneRoute>

    </zone>

</platform>
```

A.2. Workload Configurations

Listing A.3: WLCG Workload Configuration KIT

```
{
    "Analysis_T1": {
        "num_jobs": 15,
        "cores": {
            "type": "histogram",
            "counts": [0, 206746, 35, 0, 3709, 0, 0, 0, 406],
            "bins": [-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5]
        },
        "flops": {
            "type": "histogram",
            "counts": [210623, 263, 4, 6, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [650, 204277, 3161, 1560, 20, 0, 949, 0, 225, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },

        "outfilesize": {
            "type": "histogram",
            "counts": [210319, 577, 0, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Analysis_T1"
    },
    "Digi_T1": {
        "num_jobs": 15,
        "cores": {
            "type": "histogram",
            "counts": [0, 30704, 101899, 0, 86729, 0, 0, 0, 626]
        },
        "flops": {
            "type": "histogram",
            "counts": [74119, 141559, 3664, 38, 213, 317, 40, 8, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [131371, 8158, 54096, 20739, 192, 0, 5398, 0, 4, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type": "histogram",
            "counts": [155865, 59084, 1274, 593, 280, 211, 383, 131, 1528, 255],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Digi_T1"
    },
    "DataProcessing_T1": {
        "num_jobs": 10,
        "cores": {
            "type": "histogram",
            "counts": [0, 0, 1776, 0, 54222, 0, 0, 0, 0]
        },
        "flops": {
            "type":"histogram",
            "counts": [21063, 5534, 6066, 6909, 6006, 5984, 3002, 1030, 322, 57],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [295, 1339, 182, 200, 2095, 0, 50907, 0, 980, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [53731, 1799, 61, 231, 166, 10, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "DataProcessing_T1"
    },
    "Others_T1": {
        "num_jobs": 5,
        "cores": {
            "type": "histogram",
            "counts": [0, 0, 28, 0, 485, 0, 0, 0, 282]
        },
        "flops": {
            "type":"histogram",
            "counts": [789, 2, 0, 4, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [38, 270, 477, 10, 0, 0, 0, 0, 0, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [612, 175, 4, 0, 4, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+09, 3e+09, 4.5e+09, 6e+09, 7.5e+09, 9e+09, 1.05e+10,
                1.2e+10, 1.35e+10, 1.5e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Others_T1"
    },
    "Merge_T1": {
        "num_jobs": 5,
        "cores": {
            "type": "histogram",
            "counts": [0, 23054, 0, 0, 0, 0, 0, 0, 0]
        },
        "flops": {
            "type":"histogram",
            "counts": [23054, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [16533, 6521, 0, 0, 0, 0, 0, 0, 0, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [3170, 15334, 4550, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Merge_T1"
    }
}
```

Listing A.4: WLCG Workload Configuration KIT and DESY

```
{
    "Analysis_T2": {
        "num_jobs": 15,
        "cores": {
            "type": "histogram",
            "counts": [0, 441598, 10, 0, 29325, 0, 0, 0, 5244],
            "bins": [-0.5, 0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5]
        },
        "flops": {
            "type": "histogram",
            "counts": [471947, 4169, 54, 2, 0, 0, 5, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [644, 271820, 86090, 82679, 8262, 0, 22801, 0, 1764, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "infilesize": {
            "type": "histogram",
            "counts": [456792, 10442, 1316, 6796, 581, 121, 91, 0, 19, 19],
            "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e
                +10, 4.5e+10, 5e+10]
        },
        "outfilesize": {
            "type": "histogram",
            "counts": [473812, 2317, 48, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Analysis_T2"
    },
    "Digi_T2": {
        "num_jobs": 15,
        "cores": {
            "type": "histogram",
            "counts": [0, 86939, 151890, 0, 87586, 0, 0, 0, 304]
        },
        "flops": {
            "type": "histogram",
            "counts": [175687, 149681, 804, 92, 279, 156, 12, 6, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type": "histogram",
            "counts": [212797, 32272, 12499, 63662, 58, 0, 5431, 0, 0, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type": "histogram",
            "counts": [255261, 66445, 1547, 670, 349, 293, 564, 245, 889, 2],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Digi_T2"
    },
    "DataProcessing_T2": {
        "num_jobs": 10,
        "cores": {
            "type": "histogram",
            "counts": [0, 0, 1613, 0, 47233, 0, 0, 0, 0]
        },
        "flops": {
            "type":"histogram",
            "counts": [12823, 3957, 13415, 10139, 6275, 1490, 499, 144, 61, 23],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [76, 1499, 42, 113, 5541, 0, 41522, 0, 53, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [47160, 1623, 51, 2, 8, 2, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "DataProcessing_T2"
    },
    "Others_T2": {
        "num_jobs": 5,
        "cores": {
            "type": "histogram",
            "counts": [0, 0, 127, 0, 998, 0, 0, 0, 608]
        },
        "flops": {
            "type":"histogram",
            "counts": [1697, 34, 2, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [50, 263, 1416, 4, 0, 0, 0, 0, 0, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [1402, 313, 14, 2, 0, 2, 0, 0, 0, 0],
            "bins": [0, 1.5e+09, 3e+09, 4.5e+09, 6e+09, 7.5e+09, 9e+09, 1.05e+10,
                1.2e+10, 1.35e+10, 1.5e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Others_T2"
    },
    "Merge_T2": {
        "num_jobs": 5,
        "cores": {
            "type": "histogram",
            "counts": [0, 18627, 0, 0, 0, 0, 0, 0, 0]
        },
        "flops": {
            "type":"histogram",
            "counts": [18627, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 1.5e+14, 3e+14, 4.5e+14, 6e+14, 7.5e+14, 9e+14, 1.05e+15,
                1.2e+15, 1.35e+15, 1.5e+15]
        },
        "memory": {
            "type":"histogram",
            "counts": [7877, 10746, 4, 0, 0, 0, 0, 0, 0, 0],
            "bins": [100, 1290, 2480, 3670, 4860, 6050, 7240, 8430, 9620, 10810,
                12000]
        },
        "outfilesize": {
            "type":"histogram",
            "counts": [3677, 11877, 3073, 0, 0, 0, 0, 0, 0, 0],
            "bins": [0, 2e+09, 4e+09, 6e+09, 8e+09, 1e+10, 1.2e+10, 1.4e+10, 1.6e
                +10, 1.8e+10, 2e+10]
        },
        "workload_type": "streaming",
        "submission_time": 10,
        "infile_datasets": "Merge_T2"
    }
}
```

A.3. Dataset Configurations

Listing A.5: WLCG Dataset Configuration for RemoteStorage

```
{
  "Analysis_T1":{
    "location":"RemoteStorage",
    "num_files":5000,
    "filesize": {
            "type": "histogram",
            "counts": [208159, 967, 701, 447, 270, 154, 72, 31, 38, 6],
            "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e
                +10, 4.5e+10, 5e+10]
    }
  },
  "Digi_T1":{
    "location":"RemoteStorage",
    "num_files":5000,
    "filesize": {
        "type": "histogram",
        "counts": [157564, 3767, 32275, 353, 25151, 89, 0, 0, 759, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  },
  "DataProcessing_T1":{
    "location":"RemoteStorage",
    "num_files":2500,
    "filesize": {
        "type":"histogram",
        "counts": [54197, 1801, 0, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  },
  "Others_T1":{
    "location":"RemoteStorage",
    "num_files":2500,
    "filesize": {
        "type":"histogram",
        "counts": [687, 102, 6, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 3e+09, 6e+09, 9e+09, 1.2e+10, 1.5e+10, 1.8e+10, 2.1e+10, 2.4e
            +10, 2.7e+10, 3e+10]
    }
  },
  "Merge_T1":{
    "location":"RemoteStorage",
    "num_files":2500,
    "filesize": {
        "type":"histogram",
        "counts": [23054, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  }
}
```

Listing A.6: WLCG Dataset Configuration for GridKA_dCache

```
{
  "Analysis_T2":{
    "location":"GridKA_dCache",
    "num_files":500,
    "filesize": {
        "type": "histogram",
        "counts": [456792, 10442, 1316, 6796, 581, 121, 91, 0, 19, 19],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  },
  "Digi_T2":{
    "location":"GridKA_dCache",
    "num_files":500,
    "filesize": {
        "type": "histogram",
        "counts": [323934, 2694, 91, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10,
            4.5e+10, 5e+10]
    }
  },
  "DataProcessing_T2":{
    "location":"GridKA_dCache",
    "num_files":500,
    "filesize": {
        "type":"histogram",
        "counts": [48846, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10, 4.5e+10, 5e+10]
    }
  },
  "Others_T2":{
    "location":"GridKA_dCache",
    "num_files":250,
    "filesize": {
        "type":"histogram",
        "counts": [1733, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 3e+09, 6e+09, 9e+09, 1.2e+10, 1.5e+10, 1.8e+10, 2.1e+10, 2.4e+10, 2.7e+10, 3e+10]
    }
  },
  "Merge_T2":{
    "location":"GridKA_dCache",
    "num_files":250,
    "filesize": {
        "type":"histogram",
        "counts": [18627, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        "bins": [0, 5e+09, 1e+10, 1.5e+10, 2e+10, 2.5e+10, 3e+10, 3.5e+10, 4e+10, 4.5e+10, 5e+10]
    }
  }
}
```

Bibliography

[1] Md Ahsan et al. “Effect of Data Scaling Methods on Machine Learning Algorithms and Model Performance”. en. In: *Technologies* 9.3 (July 2021), p. 52. ISSN: 2227-7080. DOI: `10.3390/technologies9030052`.

[2] Prasanna Balaprakash, Robert B. Gramacy, and Stefan M. Wild. “Active-learning-based surrogate models for empirical performance tuning”. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. Indianapolis, IN, USA: IEEE, Sept. 2013, pp. 1–8. ISBN: 978-1-4799-0898-1. DOI: `10.1109/CLUSTER.2013.6702683`.

[3] S. Balasundaram and Subhash Chandra Prasad. “Robust twin support vector regression based on Huber loss function”. en. In: *Neural Computing and Applications* 32.15 (Aug. 2020), pp. 11285–11309. ISSN: 0941-0643, 1433-3058. DOI: `10.1007/s00521-019-04625-8`.

[4] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984), pp. 728–738. ISSN: 0098-5589. DOI: `10.1109/TSE.1984.5010301`.

[5] Steffen Becker, Heiko Koziolek, and Ralf Reussner. “Model-Based performance prediction with the palladio component model”. en. In: *Proceedings of the 6th international workshop on Software and performance*. Buenes Aires Argentina: ACM, Feb. 2007, pp. 54–65. ISBN: 978-1-59593-297-6. DOI: `10.1145/1216993.1217006`.

[6] I. Bird et al. *Update of the Computing Models of the WLCG and the LHC Experiments*. en. Tech. rep. Number: CERN-LHCC-2014-014. Apr. 2014.

[7] Jannis Born and Matteo Manica. “Regression Transformer enables concurrent sequence regression and generation for molecular language modelling”. en. In: *Nature Machine Intelligence* 5.4 (Apr. 2023), pp. 432–444. ISSN: 2522-5839. DOI: `10.1038/s42256-023-00639-z`.

[8] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. “Machine Learning for Fluid Mechanics”. en. In: *Annual Review of Fluid Mechanics* 52.1 (Jan. 2020), pp. 477–508. ISSN: 0066-4189, 1545-4479. DOI: `10.1146/annurev-fluid-010719-060214`.

[9] Peter Bühlmann. “Bagging, Boosting and Ensemble Methods”. en. In: *Handbook of Computational Statistics*. Ed. by James E. Gentle, Wolfgang Karl Härdle, and Yuichi Mori. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 985–1022. ISBN: 978-3-642-21550-6 978-3-642-21551-3. DOI: `10.1007/978-3-642-21551-3_33`.

[10] Rajkumar Buyya and Manzur Murshed. *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*. arXiv:cs/0203019. Mar. 2002. URL: `http://arxiv.org/abs/cs/0203019` (visited on 11/15/2023).

[11] Alexandru Calotoiu et al. “Fast Multi-parameter Performance Modeling”. In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. Taipei, Taiwan: IEEE, Sept. 2016, pp. 172–181. ISBN: 978-1-5090-3653-0. DOI: 10.1109/CLUSTER.2016.57.

[12] H. Casanova. “Simgrid: a toolkit for the simulation of application scheduling”. In: *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. Brisbane, Qld., Australia: IEEE Comput. Soc, 2001, pp. 430–437. ISBN: 978-0-7695-1010-1. DOI: 10.1109/CCGRID.2001.923223.

[13] Henri Casanova, Arnaud Legrand, and Martin Quinson. “SimGrid: A Generic Framework for Large-Scale Distributed Experiments”. In: *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*. Cambridge, UK: IEEE, 2008, pp. 126–131. ISBN: 978-0-7695-3114-4. DOI: 10.1109/UKSIM.2008.28.

[14] Henri Casanova et al. “Developing accurate and scalable simulators of production workflow management systems with WRENCH”. en. In: *Future Generation Computer Systems* 112 (Nov. 2020), pp. 162–175. ISSN: 0167739X. DOI: 10.1016/j.future.2020.05.030.

[15] Henri Casanova et al. “Versatile, scalable, and accurate simulation of distributed applications and platforms”. en. In: *Journal of Parallel and Distributed Computing* 74.10 (Oct. 2014), pp. 2899–2917. ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.06.008.

[16] René Caspart et al. “Modeling and Simulation of Load Balancing Strategies for Computing in High Energy Physics”. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti et al., p. 03027. ISSN: 2100-014X. DOI: 10.1051/epjconf/201921403027.

[17] Patricio Cerda and Gael Varoquaux. “Encoding High-Cardinality String Categorical Variables”. In: *IEEE Transactions on Knowledge and Data Engineering* 34.3 (Mar. 2022), pp. 1164–1176. ISSN: 1041-4347, 1558-2191, 2326-3865. DOI: 10.1109/TKDE.2020.2992529.

[18] CERN. “CERN Annual report 2022”. en. In: (2023). Publisher: CERN Document Server. DOI: 10.17181/ANNUALREPORT2022.

[19] T. Chai and R. R. Draxler. “Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature”. en. In: *Geoscientific Model Development* 7.3 (June 2014), pp. 1247–1250. ISSN: 1991-9603. DOI: 10.5194/gmd-7-1247-2014.

[20] Philippe Charpentier. “LHC Computing: past, present and future”. In: *EPJ Web of Conferences* 214 (2019). Ed. by A. Forti et al., p. 09009. ISSN: 2100-014X. DOI: 10.1051/epjconf/201921409009.

[21] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. arXiv:1406.1078 [cs, stat]. Sept. 2014. URL: http://arxiv.org/abs/1406.1078 (visited on 03/03/2024).

[22] A. Colin Cameron and Frank A.G. Windmeijer. “An R-squared measure of goodness of fit for some common nonlinear regression models”. en. In: *Journal of Econometrics* 77.2 (Apr. 1997), pp. 329–342. ISSN: 03044076. DOI: 10.1016/S0304-4076(96)01818-0.

[23] Wim Depoorter et al. “Scalability of Grid Simulators: An Evaluation”. en. In: *Euro-Par 2008 – Parallel Processing*. Ed. by Emilio Luque, Tomàs Margalef, and Domingo Benítez. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 544–553. ISBN: 978-3-540-85451-7. DOI: 10.1007/978-3-540-85451-7_58.

[24] Yann Dubois et al. “Location Attention for Extrapolation to Longer Sequences”. In: (2019). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.1911.03872.

[25] A Forti et al. “Multicore job scheduling in the Worldwide LHC Computing Grid”. In: *Journal of Physics: Conference Series* 664.6 (Dec. 2015), p. 062016. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/664/6/062016.

[26] John T. Hancock and Taghi M. Khoshgoftaar. “Survey on categorical data for neural networks”. en. In: *Journal of Big Data* 7.1 (Dec. 2020), p. 28. ISSN: 2196-1115. DOI: 10.1186/s40537-020-00305-w.

[27] Oliver Hennigh et al. “NVIDIA SimNet^{TM}: an AI-accelerated multi-physics simulation framework”. In: (2020). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2012.07938.

[28] Jane Hillston. *A Compositional Approach to Performance Modelling*. 1st ed. Cambridge University Press, June 1996. ISBN: 978-0-521-57189-0 978-0-521-67353-2 978-0-511-56995-1. DOI: 10.1017/CBO9780511569951.

[29] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. en. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1997.9.8.1735.

[30] Maximilian Horzela et al. *26TH INTERNATIONAL CONFERENCE ON COMPUTING IN HIGH ENERGY & NUCLEAR PHYSICS (CHEP2023)*. Aug. 2023. URL: https://indico.jlab.org/event/459/contributions/11490/ (visited on 11/03/2023).

[31] Maximilian Horzela et al. *HEPCompSim/DCSim: DCSim simulator release v0.3*. Aug. 2023. DOI: 10.5281/ZENODO.8300961. URL: https://zenodo.org/record/8300961 (visited on 10/31/2023).

[32] Reka Howard and Diego Jarquin. “Genomic Prediction Using Canopy Coverage Image and Genotypic Information in Soybean via a Hybrid Model”. en. In: *Evolutionary Bioinformatics* 15 (Jan. 2019), p. 117693431984002. ISSN: 1176-9343, 1176-9343. DOI: 10.1177/1176934319840026.

[33] Frank Hutter et al. “Algorithm runtime prediction: Methods & evaluation”. en. In: *Artificial Intelligence* 206 (Jan. 2014), pp. 79–111. ISSN: 00043702. DOI: 10.1016/j.artint.2013.10.003.

[34] *Interpolation — The Science of Machine Learning*. URL: https://www.ml-science.com/interpolation (visited on 11/10/2023).

[35] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. en. In: *Proceedings of the 2019 Conference of the North*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[36] Aryan Jadon, Avinash Patil, and Shruti Jadon. *A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting*. arXiv:2211.02989 [cs]. Nov. 2022. URL: `http://arxiv.org/abs/2211.02989` (visited on 02/09/2024).

[37] C Jung et al. "ALICE Grid Computing at the GridKa Tier-1 Center". In: *Journal of Physics: Conference Series* 396.4 (Dec. 2012), p. 042032. ISSN: 1742-6588, 1742-6596. DOI: `10.1088/1742-6596/396/4/042032`.

[38] Kamal Basulaiman and Masoud Barati. "Sequence-to-Sequence Forecasting-aided State Estimation for Power Systems". In: *2021 IEEE Texas Power and Energy Conference (TPEC)*. College Station, TX, USA: IEEE, Feb. 2021, pp. 1–6. ISBN: 978-1-72818-612-2. DOI: `10.1109/TPEC51183.2021.9384984`.

[39] Khaled A. Althelaya, El-Sayed M. El-Alfy, and Salahadin Mohammed. "Evaluation of bidirectional LSTM for short-and long-term stock market prediction". In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. Irbid: IEEE, Apr. 2018, pp. 151–156. ISBN: 978-1-5386-4366-2. DOI: `10.1109/IACS.2018.8355458`.

[40] *KIT - SCC - Research - Data Management, Data Analysis and secure IT Federations - GridKa*. 2022. URL: `https://www.scc.kit.edu/en/research/gridka.php` (visited on 03/04/2024).

[41] M. Křížek. "Brezinski, C.; Redivo Zaglia, M., Extrapolation Methods. Theory and Practice. Amsterdam etc., North-Holland 1991. X, 464 pp., Dfl. 225.00. ISBN 0-444-88814-4 (Studies in Computational Mathematics 2)". en. In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 73.9 (Jan. 1993), pp. 236–236. ISSN: 0044-2267, 1521-4001. DOI: `10.1002/zamm.19930730912`.

[42] Adrien Lebre et al. "Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Shenzhen, China: IEEE, May 2015, pp. 251–260. ISBN: 978-1-4799-8006-2. DOI: `10.1109/CCGrid.2015.134`.

[43] I.C. Legrand and H.B. Newman. "The MONARC toolset for simulating large network-distributed processing systems". In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. Vol. 2. Orlando, FL, USA: IEEE, 2000, pp. 1794–1801. ISBN: 978-0-7803-6579-7. DOI: `10.1109/WSC.2000.899171`.

[44] Petro Liashchynskyi and Pavlo Liashchynskyi. *Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS*. arXiv:1912.06059 [cs, stat]. Dec. 2019. URL: `http://arxiv.org/abs/1912.06059` (visited on 02/08/2024).

[45] Jefrey Lijffijt, Panagiotis Papapetrou, and Kai Puolamäki. "Size Matters: Finding the Most Informative Set of Window Lengths". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by David Hutchison et al. Vol. 7524. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 451–466. ISBN: 978-3-642-33485-6 978-3-642-33486-3. DOI: `10.1007/978-3-642-33486-3_29`.

[46] T Maeno et al. “Overview of ATLAS PanDA Workload Management”. In: *Journal of Physics: Conference Series* 331.7 (Dec. 2011), p. 072024. ISSN: 1742-6596. DOI: 10.1088/1742-6596/331/7/072024.

[47] Dzmitry Makatun et al. “Simulations and study of a new scheduling approach for distributed data production”. In: *Journal of Physics: Conference Series* 762 (Oct. 2016), p. 012023. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/762/1/012023.

[48] Preeti Malakar et al. “Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications”. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS).* Dallas, TX, USA: IEEE, Nov. 2018, pp. 33–44. ISBN: 978-1-72810-182-8. DOI: 10.1109/PMBS.2018.8641686.

[49] *Mean Squared Error (MSE).* URL: https://www.probabilitycourse.com/chapter9/9_1_5_mean_squared_error_MSE.php (visited on 11/06/2023).

[50] L. R. Medsker and L. C. Jain, eds. *Recurrent neural networks: design and applications.* The CRC Press international series on computational intelligence. Boca Raton, FL: CRC Press, 2000. ISBN: 978-0-8493-7181-3.

[51] Luke Merrick. *Randomized Ablation Feature Importance.* arXiv:1910.00174 [cs, stat]. Oct. 2019. URL: http://arxiv.org/abs/1910.00174 (visited on 04/28/2024).

[52] Siddartha Mootha et al. “Stock Price Prediction using Bi-Directional LSTM based Sequence to Sequence Modeling and Multitask Learning”. In: *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON).* New York, NY, USA: IEEE, Oct. 2020, pp. 0078–0086. ISBN: 978-1-72819-656-5. DOI: 10.1109/UEMCON51285.2020.9298066.

[53] Haykuhi Musheghyan et al. “The GridKa tape storage: latest improvements and current production setup”. en. In: (2021). Publisher: EDP Sciences. DOI: 10.5445/IR/1000154088.

[54] Deepak Narayanan et al. *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM.* arXiv:2104.04473 [cs]. Aug. 2021. URL: http://arxiv.org/abs/2104.04473 (visited on 03/03/2024).

[55] Harvey B. Newman. “Simulating distributed systems”. en. In: *AIP Conference Proceedings.* Vol. 583. ISSN: 0094243X. Batavia, Illinois (USA): AIP, 2001, pp. 164–166. DOI: 10.1063/1.1405293.

[56] Truong Thao Nguyen et al. “Why Globally Re-shuffle? Revisiting Data Shuffling in Large Scale Deep Learning”. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* Lyon, France: IEEE, May 2022, pp. 1085–1096. ISBN: 978-1-66548-106-9. DOI: 10.1109/IPDPS53621.2022.00109.

[57] Seol-Hyun Noh. “Analysis of Gradient Vanishing of RNNs and Performance Comparison”. en. In: *Information* 12.11 (Oct. 2021), p. 442. ISSN: 2078-2489. DOI: 10.3390/info12110442.

[58] Peshawa J Muhammad Ali and Rezhna Hassan Faraj. “Data Normalization and Standardization: A Technical Report”. en. In: (2014). Publisher: Unpublished. DOI: 10.13140/RG.2.2.28948.04489.

[59] Jun Qi et al. “On Mean Absolute Error for Deep Neural Network Based Vector-to-Vector Regression”. In: *IEEE Signal Processing Letters* 27 (2020), pp. 1485–1489. ISSN: 1070-9908, 1558-2361. DOI: 10.1109/LSP.2020.3016837.

[60] Antônio H. Ribeiro et al. *Beyond exploding and vanishing gradients: analysing RNN training using attractors and smoothness*. arXiv:1906.08482 [cs, math, stat]. Mar. 2020. URL: http://arxiv.org/abs/1906.08482 (visited on 03/03/2024).

[61] Hojjat Salehinejad et al. “Recent Advances in Recurrent Neural Networks”. In: (2018). Publisher: arXiv Version Number: 3. DOI: 10.48550/ARXIV.1801.01078.

[62] Sreelekshmy Selvin et al. “Stock price prediction using LSTM, RNN and CNN-sliding window model”. In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Udupi: IEEE, Sept. 2017, pp. 1643–1647. ISBN: 978-1-5090-6367-3. DOI: 10.1109/ICACCI.2017.8126078.

[63] Akhil Sethia and Purva Raut. “Application of LSTM, GRU and ICA for Stock Price Prediction”. In: *Information and Communication Technology for Intelligent Systems*. Ed. by Suresh Chandra Satapathy and Amit Joshi. Vol. 107. Series Title: Smart Innovation, Systems and Technologies. Singapore: Springer Singapore, 2019, pp. 479–487. ISBN: 9789811317460 9789811317477. DOI: 10.1007/978-981-13-1747-7_46.

[64] Jamie Shiers. “The Worldwide LHC Computing Grid (worldwide LCG)”. en. In: *Computer Physics Communications* 177.1-2 (July 2007), pp. 219–223. ISSN: 00104655. DOI: 10.1016/j.cpc.2007.02.021.

[65] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. “The Performance of LSTM and BiLSTM in Forecasting Time Series”. In: *2019 IEEE International Conference on Big Data (Big Data)*. Los Angeles, CA, USA: IEEE, Dec. 2019, pp. 3285–3292. ISBN: 978-1-72810-858-2. DOI: 10.1109/BigData47090.2019.9005997.

[66] B.W. Silverman. *Density Estimation for Statistics and Data Analysis*. en. 1st ed. Routledge, Feb. 2018. ISBN: 978-1-315-14091-9. DOI: 10.1201/9781315140919.

[67] *sklearn.preprocessing.StandardScaler — scikit-learn 1.4.1 documentation*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html (visited on 03/10/2024).

[68] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. arXiv:1803.09820 [cs, stat]. Apr. 2018. URL: http://arxiv.org/abs/1803.09820 (visited on 05/05/2024).

[69] Anthony Sulistio et al. “A toolkit for modelling and simulating data Grids: an extension to GridSim”. en. In: *Concurrency and Computation: Practice and Experience* 20.13 (Sept. 2008), pp. 1591–1609. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.1307.

[70] Jingwei Sun et al. “Automated Performance Modeling of HPC Applications Using Machine Learning”. In: *IEEE Transactions on Computers* 69.5 (May 2020), pp. 749–763. ISSN: 0018-9340, 1557-9956, 2326-3814. DOI: 10.1109/TC.2020.2964767.

[71] Ashish Vaswani et al. “Attention Is All You Need”. In: (2017). Publisher: arXiv Version Number: 7. DOI: 10.48550/ARXIV.1706.03762.

[72] Borislava Vrigazova. “The Proportion for Splitting Data into Training and Test Set for the Bootstrap in Classification Problems”. en. In: *Business Systems Research Journal* 12.1 (May 2021), pp. 228–242. ISSN: 1847-9375. DOI: 10.2478/bsrj-2021-0015.

[73] Shuhei Watanabe. *Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance.* arXiv:2304.11127 [cs]. May 2023. URL: http://arxiv.org/abs/2304.11127 (visited on 02/08/2024).

[74] Bob Wescott. *The Every Computer Performance Book: How to avoid and solve performance problems on the computer you work with.* eng. First ed. Leipzig: Amazon Distribution [Print On Demand], 2013. ISBN: 978-1-4826-5775-3.

[75] Cj Willmott and K Matsuura. “Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance”. en. In: *Climate Research* 30 (2005), pp. 79–82. ISSN: 0936-577X, 1616-1572. DOI: 10.3354/cr030079.

[76] Carl Witt et al. “Predictive performance modeling for distributed batch processing using black box monitoring and machine learning”. en. In: *Information Systems* 82 (May 2019), pp. 33–52. ISSN: 03064379. DOI: 10.1016/j.is.2019.01.006.

[77] Neo Wu et al. “Deep Transformer Models for Time Series Forecasting: The Influenza Prevalence Case”. In: (2020). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2001.08317.

[78] Peter T. Yamak, Li Yujian, and Pius K. Gadosey. “A Comparison between ARIMA, LSTM, and GRU for Time Series Forecasting”. en. In: *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence.* Sanya China: ACM, Dec. 2019, pp. 49–55. ISBN: 978-1-4503-7261-9. DOI: 10.1145/3377713.3377722.

[79] Li Yang and Abdallah Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. en. In: *Neurocomputing* 415 (Nov. 2020), pp. 295–316. ISSN: 09252312. DOI: 10.1016/j.neucom.2020.07.061.

[80] Yuanshuai Duan et al. “Improved BIGRU Model and Its Application in Stock Price Forecasting”. en. In: *Electronics* 12.12 (June 2023), p. 2718. ISSN: 2079-9292. DOI: 10.3390/electronics12122718.

[81] Jieyu Zhang et al. *WRENCH: A Comprehensive Benchmark for Weak Supervision.* arXiv:2109.11377 [cs, stat]. Oct. 2021. URL: http://arxiv.org/abs/2109.11377 (visited on 10/31/2023).