Big Data Analysis - Deucalion Log's

Augusto Campos
Universidade do Minho
Braga, Portugal
PG57510@uminho.pt

Carlos Silva
Universidade do Minho
Braga, Portugal
PG57518@uminho.pt

Tiago P. Silva
Universidade do Minho
Braga, Portugal
PG57617@uminho.pt

Figure 1: The Deucalion Super Computer.

Abstract

This work addresses the challenge of predicting job failures in a high-performance computing (HPC) environment using large-scale system and job execution logs from the Deucalion supercomputer. We present an end-to-end data processing and machine learning pipeline that leverages Apache Spark for scalable log preprocessing and feature engineering, followed by distributed deep learning model training using PyTorch Tabular's TabNet architecture. Our preprocessing phase includes correcting structural issues in raw logs. The Spark stage focuses on efficient dataset construction by expanding node information, extracting temporal features, joining system logs with job metadata, and optimizing cluster resource utilization through query optimization. The PyTorch stage implements distributed training with a focus on balancing batch size and node count, although constrained by limited training time. We discuss challenges related to distributed optimization, potential benefits of early stopping, and considerations for optimizer selection. Our results demonstrate the feasibility and challenges of building scalable HPC failure prediction systems and highlight avenues for future research and optimization.

CCS Concepts

• **Computing methodologies** → **MapReduce algorithms**; Classification and regression trees; *Cooperation and coordination.*

Keywords

High-Performance Computing, Job Failure Prediction, Apache Spark, Distributed Deep Learning, PyTorch Tabular, TabNet, Data Preprocessing, Cluster Scalability, Machine Learning, Log Analysis

1 Introduction

The increasing complexity and scale of High-Performance Computing (HPC) environments pose significant challenges to maintaining system reliability and performance. In particular, anticipating job failures based on system logs and job execution metadata has become an essential task to reduce waste of computational resources and improve scheduling efficiency.

In this project, we address the problem of job failure prediction using logs and execution data collected from a supercomputing infrastructure (Deucalion). The dataset, provided as part of the practical assignment, includes `syslog` and `slurm` logs from January 31 to March 16, 2025. The main goal is to build an end-to-end pipeline that preprocesses these logs using Apache Spark and trains a deep learning model using distributed PyTorch.

Before reaching the Spark stage, we apply a preprocessing phase using the **AWK** programming language. This step is essential to repair structural inconsistencies in the raw logs, such as missing delimiters and others. AWK was chosen for its high performance in pattern-based text manipulation, which is particularly suited to large-scale log parsing tasks.

Once the log structure is corrected, the data is ingested into Apache Spark, where further parsing, enrichment, and feature engineering are applied. These include the extraction of temporal attributes (e.g., day of the week, hour, minute), job metadata (e.g., job ID, CPU hours, number of nodes), and categorical indicators (e.g., partition, host). The preprocessed data is stored in Parquet format to enable efficient access and scalability.

Our implementation adopts **TabNet**, a deep learning architecture for tabular data, accessed through the `pytorch-tabular` library. We chose this model with the help of the teaching team, its interpretability and performance on heterogeneous features, including categorical and continuous variables, which are part of our dataset.

The pipeline is deployed and tested on a distributed cluster, using Apache Spark for the data processing stages and the `pytorch-tabular TabularModel` for training the model across multiple nodes. The entire workflow is designed to be automated and modular, allowing for reproducibility and experimental benchmarking.

In this article, we present the architecture of our solution, the choices made during its implementation, and an evaluation of its performance under different configurations. We also provide a discussion of the challenges encountered, lessons learned, and directions for future work.

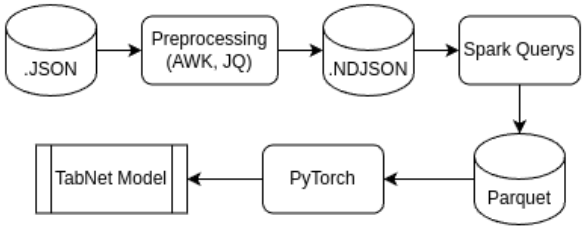


Figure 2: Pipeline Architecture.

2 Preprocessing Stage

2.1 Addressing Structural Issues

The data provided to us presented some structural errors that made it impossible to consume using the Apache Spark framework.

The data is basically composed of two different types of files: the first consists of system logs for a given day between 31st January and 16th March, and the second is a single file containing all Slurm job executions for the entire period.

The first error encountered, which prevented parsing the JSONs using Spark, was that each logstash file was composed of individual JSON arrays of 1000 items each. When ingesting the data with Spark, it could only consume the first array and ignored the rest of the data.

The second error was that the last entry in every logstash was cut in half, meaning it was not a complete JSON entry.

Moreover, the logstash file from 7th March had a different schema from all the others; it was missing information that was vital to our final model and would have introduced a large number of null values in the features. For this reason, we excluded the logstash from that day.

The first and second errors were fixed using the `tac` bash command (to reverse the input order of the file) in conjunction with AWK, as mentioned before. For each file, we launched a Slurm job that fixed the structure of the corresponding day, effectively parallelising this step.

2.2 Spark Ingestion Format

After this, we obtained a single JSON array of objects per day (each amounting to several GBs), and a single array for the Slurm job information, which did not suffer from the issues mentioned previously.

When consuming these files in Spark, query execution times were significantly longer, especially when the number of nodes was small. This led us to theorise that reading a large array was not an optimal input format for Spark. We then tested the reading of a single logstash file to observe how Spark distributed the array across its nodes. What we observed was that when reading a single file containing one large array, Spark only used one executor for that stage. This clearly represented a bottleneck: even if we had one executor per file, this setup did not allow for dynamic distribution of data to free executors (some files were around 1 GB, others around 7 GB).

Table 1: Comparison of times with and without jq in seconds.

Day	Time with jq	Time without jq
2025.01.31	1008	229
2025.02.01	227	51
2025.02.01	228	51
2025.02.02	225	51
2025.02.03	230	52
2025.02.04	231	52
2025.02.05	876	201
2025.02.06	868	197
2025.02.07	228	52
2025.02.08	926	212
2025.02.09	862	199
2025.02.10	875	696
2025.02.11	230	52
2025.02.12	945	217
2025.02.13	1062	240
2025.02.14	1357	303
2025.02.15	229	53
2025.02.16	938	215
2025.02.17	229	53
2025.02.18	1151	265
2025.02.19	230	54
2025.02.20	1220	281
2025.02.21	1069	247
2025.02.22	231	53
2025.02.23	231	52
2025.02.24	1303	293
2025.02.25	226	53
2025.02.26	244	51
2025.02.27	1327	325
2025.02.28	1323	295
2025.03.01	1369	305
2025.03.02	1281	288
2025.03.03	1289	291
2025.03.04	344	170
2025.03.05	1178	266
2025.03.06	1265	287
2025.03.07	45	46
2025.03.08	1298	287
2025.03.09	220	48
2025.03.10	236	50
2025.03.11	1314	297
2025.03.12	1234	276
2025.03.13	1294	303
2025.03.14	1360	294
2025.03.15	1411	298
2025.03.16	1395	295

We then used the `jq` bash command as part of our preprocessing stage, transforming the large JSON array into newline-delimited JSON (NDJSON) format. When consumed by Spark, this significantly reduced the query execution time.

Although Spark queries became faster, the time added by using `jq` was itself a pipeline bottleneck, larger files added several minutes of preprocessing using this command-line utility.

We ultimately decided to adopt the version with `jq` in NDJSON format because the subsequent optimisation and iterative development of Spark queries was more dynamic. Without it, we wouldn't have had a consistent workflow. We also considered that the objective of this practical assignment was to optimise the Spark and PyTorch cluster configurations, giving less importance to the step of fixing the log structure. Ideally, this data would be extracted from Deucalion already in NDJSON format, which would eliminate the need for this entire stage and allow the pipeline to begin directly with Spark.

In later stages, we also considered the option of having multiple arrays per day split into different files, which we thought might be better handled by Spark than a single array. However, due to time constraints, we did not test this option.

3 Apache Spark Stage

In this stage of the pipeline, we constructed the final dataset that served as the foundation for our model training.

3.1 Node Expansion

The first step involved loading data from the SLURM log file. This file contains detailed information about jobs submitted to the system, and is essential for the classification task.

The `nodes` field in this data included several different formats:

- `gnx529`: A single node.
- `cna[0062,0064]`: A list of specific nodes.
- `cna[0874-0921]`: A range of nodes.
- `cna[052-059,0062,098-099]`: A combination of all.

On the other hand, the system logs (logstash) contain only individual node names, like in the first format. Therefore, it was necessary to expand the `nodes` field from the SLURM JSON, and for each job, create one entry per node associated with it.

This was accomplished using regular expressions to parse and expand the `nodes` string. To apply this transformation to the Spark DataFrame, the function that uses regex was registered as a User Defined Function (UDF).

The array with every node is then exploded using the explode function from Spark.

This step was crucial for enabling the subsequent join with the system logs (`syslogs`). Without the explicit expansion of the node lists, it would not have been possible to correctly map each job to its corresponding system log entries. The quality and granularity of this mapping had a direct impact on the quality of the final dataset and, consequently, on the performance of the machine learning model.

3.2 Feature Extraction

We then proceeded to read the logstash files for each day and extracted temporal features from the `timestamp_log` field. A string containing both date and time information, in its raw form, is not particularly useful for classification tasks. However, by decomposing this timestamp into individual components (such as year, month, day, day of the week, hour, minute, and second) we are able to derive features that may reveal temporal patterns associated with job failures. For instance, it is possible that jobs are more likely to fail during specific hours of the day or days of the week. These extracted features can therefore provide valuable signals to the model, improving its ability to learn such patterns.

3.3 Joining Datasets

Following this, we performed an inner join between the two dataframes: one containing a row for each individual node associated with a SLURM job, and the other comprising system log entries for individual nodes. The join was carried out using the node identifier as the key, ensuring that only entries corresponding to the same node in both datasets were retained.

Before performing the join, we applied a `repartition` operation to both DataFrames and explicitly broadcasted the SLURM DataFrame. In Apache Spark, joins are considered *wide transformations*, meaning that they require data from multiple partitions to be shuffled across the cluster. This data movement can significantly degrade performance, especially when the datasets involved are large or imbalanced. By using `repartition`, we aimed to ensure a more uniform distribution of data across partitions, mitigating the risk of data skew and reducing contention on specific executors. Furthermore, since the SLURM DataFrame is relatively small in comparison to the syslogs dataset, we used `broadcast` to replicate it to all worker nodes. This avoids the need to shuffle the larger syslogs DataFrame, thereby converting what would be a costly shuffle join into a more efficient broadcast join. These optimizations were essential to maintain the performance and scalability of our data processing pipeline.

However, simply matching the nodes in both DataFrames was not sufficient to correctly associate a syslog entry with its corresponding SLURM job. In a typical high-performance computing environment, multiple jobs may be executed on the same node across different time intervals. Therefore, it was necessary to incorporate temporal logic into the join condition.

To address this, we leveraged the `timestamp_start` and `timestamp_end` fields from the SLURM DataFrame, which represent the execution interval of each job. We then filtered the joined dataset to retain only those syslog entries where the `timestamp_log` (the time the log event occurred) falls within the interval defined by these two timestamps. In other words, we ensured that the log entry occurred during the execution of the job on the given node.

This additional filtering step acts as a temporal join and was crucial to ensure semantic correctness in the mapping between syslogs and jobs. Without it, we would risk associating log events with the wrong job instances, which would severely compromise the quality of the resulting dataset and the accuracy of the machine learning model trained on it.

3.4 Label Encoding and Data Output

The final step in the Spark processing stage consisted of applying label encoding to a set of categorical features, namely `severity`, `state`, `exit_code`, and `qos`. This transformation was performed within the Spark environment rather than delegating it to the PyTorch cluster, due to Spark's superior performance in handling large-scale data transformations in a distributed setting.

To implement the encoding, we used Spark's `StringIndexer`, a built-in transformer that maps each distinct category in a column to a unique numerical index. In order to reduce the computational load and improve the performance of this operation, we trained each `StringIndexer` model on a random sample of the full dataset. This sample was chosen to be large enough to represent the categorical variability while avoiding the overhead of scanning the full dataset.

Label encoding is particularly well suited for classification tasks, as it transforms string-based categorical features into numerical values that deep learning models can directly interpret. Performing this step in Spark allowed us to deliver a fully prepared dataset to the PyTorch stage, thereby streamlining the end-to-end pipeline.

Finally, we wrote the final dataset in the Parquet format. Parquet is a fast, parallelizable format for writing and reading data. This choice was beneficial because the Spark cluster could write it faster, and the PyTorch cluster, which would read the data afterward, could also read it more efficiently. Parquet's columnar storage format and support for efficient compression and encoding schemes make it an ideal choice for our pipeline.

3.5 Performance and Scalability Analysis

To assess the performance of the distributed execution of our query in the Spark cluster, we conducted a speed-up analysis. Theoretically, doubling the number of nodes in the cluster should halve the execution time of the query. However, this assumes ideal conditions where computation is the only bottleneck. In practice, Spark clusters may face several other limitations, such as:

- **I/O bottlenecks:** Reading and writing large datasets from distributed storage systems can saturate bandwidth or disk throughput.
- **Shuffle overhead:** Wide transformations such as `join`, `groupBy` or `reduceByKey` require data to be shuffled across the network, which introduces latency and load on the cluster.
- **Skewed data distribution:** If the data is not evenly distributed across partitions, some executors may receive more work than others, leading to under-utilization of the cluster.
- **Task scheduling delays:** As the number of nodes increases, so does the complexity of scheduling and coordination by the driver node.

The closer our observed execution time is to the theoretical ideal, the more efficiently the cluster is utilizing its available resources. The speed-up curve, therefore, serves as an important diagnostic for the parallel scalability of the Spark stage in our pipeline.

Nodes	Mean (s)	Std Dev (s)	Min (s)	Max (s)	Runs
2	953.99	104.90	817.34	1273.26	20
4	530.94	66.32	435.69	676.25	19
8	363.92	37.06	304.65	439.24	19
16	222.98	18.69	198.54	274.44	20
32	159.74	12.68	145.59	190.58	20
64	138.40	3.14	132.42	146.09	20
128	179.79	23.59	149.71	233.66	15

Table 2: Execution time statistics for Spark query across different cluster sizes.

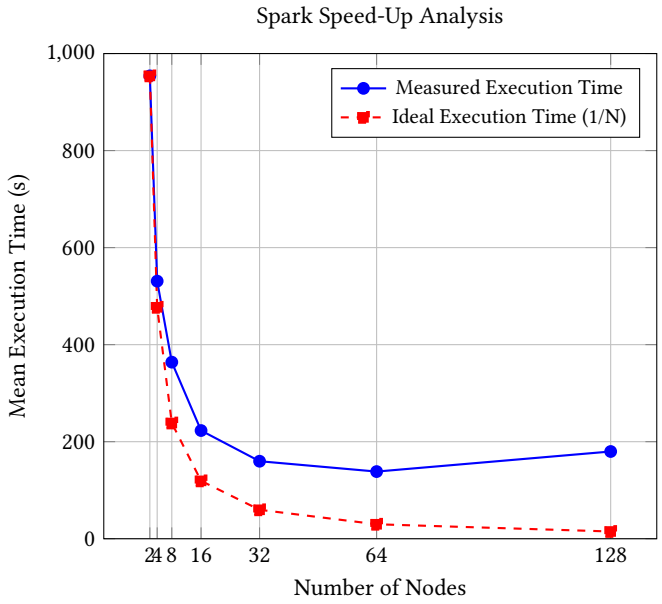


Figure 3: Comparison between measured and theoretical (ideal) execution time for Spark query.

The speed-up analysis of the Spark query execution time reveals a clear pattern of scalability up to 32 nodes, where increasing the number of nodes significantly reduces the query runtime. From 2 to 32 nodes, the performance improvement is notable and roughly follows the expected inverse relationship between execution time and the number of nodes. However, beyond 32 nodes, the gains become marginal; at 64 nodes, the decrease in execution time is much smaller, indicating diminishing returns. When scaling up to 128 nodes, the performance degrades compared to the 64-node configuration, showing an increase in query runtime.

This phenomenon can be attributed to several factors inherent to distributed systems like Spark. First, as mentioned before, the overhead of communication and coordination among a large number of executors grows with the cluster size, which can negate the benefits of parallelization. Second, Spark's shuffling and network I/O operations can become bottlenecks at high parallelism levels, leading to increased latency. Third, resource contention and task

scheduling overhead may also increase, especially if the workload does not partition evenly or if there is data skew.

3.6 Repartitioning Experiment

In order to verify whether the observed performance degradation was due to suboptimal data partitioning, we conducted an additional experiment on the 128-node configuration. In this test, the dataset was repartitioned into ten times more partitions than usual, with the objective of ensuring a more balanced distribution of data across the executors. This approach aimed to assess if improved data repartitioning could mitigate the slowdown and better utilize the available cluster resources.

Table 3: Performance metrics for the 128-node configuration with increased repartitioning

Nodes	Mean (s)	Std Dev (s)	Min (s)	Max (s)	Runs
128	228.41	23.58	184.29	263.88	12

We can see that the time gets worse with more repartition the limited parallelizability of the specific query and data size can impose a ceiling on achievable speed-up, as some stages may be inherently sequential or require synchronization.

Overall, the results emphasize the importance of carefully balancing cluster size and workload characteristics to maximize efficiency, as simply adding more nodes does not guarantee proportional improvements in performance.

4 PyTorch Stage

The next stage involves training the deep learning model using the PyTorch framework. For this task, we selected the TabNet model, implemented via the `pytorch-tabular` package.

The final dataset, obtained from the Spark processing stage and stored in Parquet format, is loaded into the PyTorch environment where it is used to train the model within a distributed architecture. This setup is essential to accelerate training by leveraging multiple nodes available in the cluster.

For training, we employed the *Distributed Data Parallel* (DDP) strategy, which enables the distribution of both the model and data across multiple devices. DDP synchronizes gradients among the different instances of the model during backpropagation, ensuring parameter consistency and efficient utilization of computational resources.

The choice of TabNet is motivated by its architecture, specifically designed for tabular data. It combines deep learning with attention mechanisms to dynamically select the most relevant features during training. This not only enhances the interpretability of the model but also provides robust performance in classification tasks such as predicting failures in HPC jobs.

During the training process, our intention was to monitor the evolution of the loss function and performance metrics, primarily accuracy, to ensure proper model convergence. To this end, we integrated tools such as TensorBoard, which offers real-time visualization of metrics, parameter histograms, and model graphs, enabling a detailed analysis of the training behavior.

However, for this stage of the pipeline, the data available on training runs was quite limited. Our group was only able to bring the TabNet model to a functional state close to the project deadline. Consequently, during the hyperparameter tuning process, and prior to initiating our speed-up analysis for the PyTorch cluster, the allotted time for the students was exhausted. Therefore, we proceed to describe the expected outcomes and evaluate the runs we were able to complete, despite suboptimal optimizations and inconsistent parameter settings, which complicate direct comparisons between them.

4.1 Loss Curve analysis

The training loss curves in Figure 4 for different configurations reveal interesting trade-offs between batch size, number of nodes, and training efficiency. Runs with larger batch sizes (e.g., 4096) exhibited faster convergence and achieved lower training loss values. Also configurations utilizing more nodes (16) with smaller batch sizes showed quicker training times but slower convergence.

This behavior likely results from the increased computational resources, which grows with the number of nodes and can increase the benefits of parallelism. Additionally, larger batch sizes provide more stable gradient estimates, promoting faster convergence but require more computational resources per iteration.

These results underscore the importance of balancing cluster size and batch size to optimize both training time and model convergence. Although distributed architectures can significantly accelerate training, inefficient scaling or improper batch size tuning can limit performance gains.

Overall, our observations align with established distributed training challenges, including synchronization costs, communication latency, and diminishing returns at scale, all of which must be carefully managed to maximize resource utilization and model quality.

Due to the limited time and computational resources available, we were only able to record a single accuracy value during the training process. This measurement was taken at the end of the first epoch from a run using 2 nodes, yielding an accuracy of approximately 0.9452. While this initial result is promising and suggests that the TabNet model is capable of effectively learning from the dataset, it is insufficient to fully assess the model's generalization performance or convergence behavior over multiple epochs. Future work should aim to conduct more comprehensive training runs with extended epochs and varied cluster sizes to better evaluate the model's effectiveness and scalability in the HPC failure prediction context.

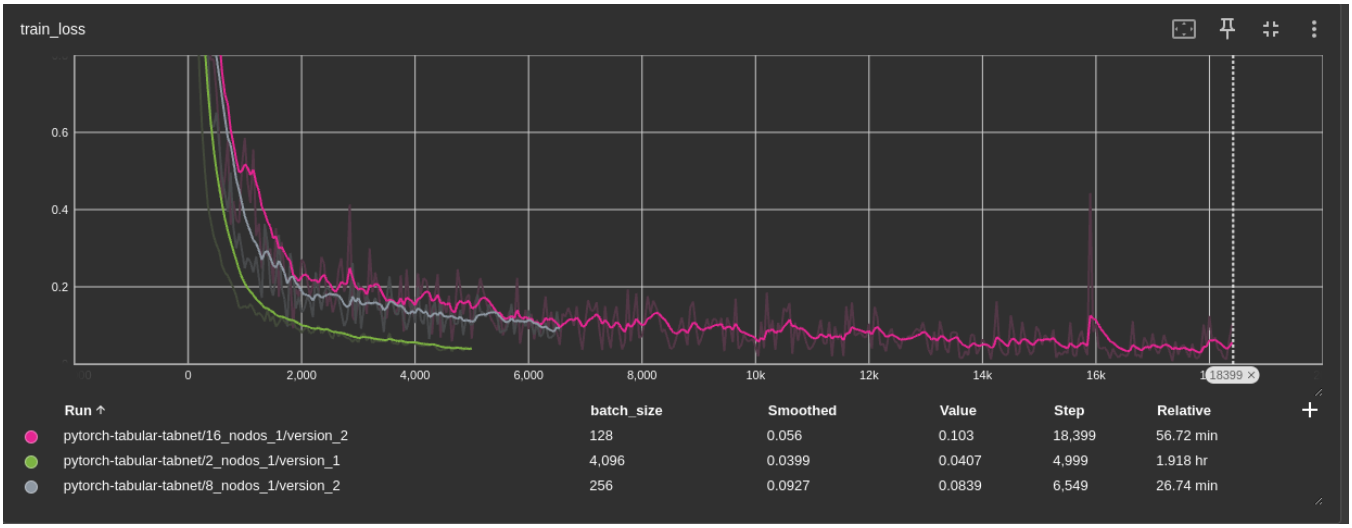
4.2 Early Stopping for Resource Optimization

Our intention was to employ the *early stopping* mechanism available in the `pytorch-tabular` framework to optimize computational resource usage and avoid overfitting during training. Early stopping monitors a specified loss or performance metric and halts training when no significant improvement is observed over a predefined number of epochs.

The parameters planned for configuration were:

- **early_stopping**: To monitor the validation loss ('valid_loss').
- **early_stopping_min_delta**: Set to 0.001, defining the minimum change to qualify as an improvement.

Figure 4: Training loss curves for 3 configurations of TabNet with different batch sizes and node counts.



- **early_stopping_mode**: Set to 'min', meaning training would stop if the monitored loss ceased decreasing.
- **early_stopping_patience**: Set to 3 epochs, defining how long to wait for improvement before stopping.

Unfortunately, due to exhaustion of the allocated training minutes on the cluster, we were unable to fully implement and evaluate early stopping in our training runs. However, this technique remains a promising approach for future work to reduce unnecessary computation and to better analyze the point at which the model has or has not converged.

4.3 Optimizer Configuration

In our implementation, we utilized the `pytorch-tabular` library, which by default employs the Adam (Adaptive Moment Estimation) optimizer from the `torch.optim` package. Adam is a widely adopted optimization algorithm in deep learning due to its adaptive learning rate mechanism, which generally provides good convergence behavior across a variety of tasks and datasets. The default learning rate used by `pytorch-tabular` is set to 1×10^{-3}, a commonly recommended starting point.

The optimizer plays a critical role at the heart of the gradient descent process, directly influencing the model's training efficiency and final performance. Although Adam is a robust choice for many applications, in a distributed training context, tuning the optimizer or switching to a different algorithm may help address challenges related to convergence stability, communication overhead, and scalability. For instance, SGD with momentum or AdamW might offer better generalization or faster convergence under certain workloads or cluster architectures.

Recent advances have already been used in DeepSpeed [5] such as 1-bit Adam, 0/1 Adam, and 1-bit LAMB optimizers have demonstrated the ability to reduce communication volume by up to 26 times compared to standard Adam, while achieving similar convergence efficiency. These quantized or compressed optimizers are particularly valuable in distributed training settings where communication overhead between nodes can become a bottleneck.

4.4 Cluster Communication and Network Considerations

The current cluster setup employs TCP protocol for communication between nodes. While TCP provided a quick setup as we had already used it in the pratical classes, it is not necessarily the most efficient protocol for high-performance computing clusters.

An alternative, such as InfiniBand, could offer significantly faster inter-node communication with lower latency and higher bandwidth. Utilizing InfiniBand technology would likely reduce communication overhead, potentially improving overall cluster performance.

However, since we have not performed a comprehensive scalability analysis (such as a speedup study) to determine the communication bottlenecks and the maximum achievable performance, the current TCP-based configuration was maintained. Consequently, this optimization was not pursued in the present work, as the cluster performance did not reach the thresholds where such improvements would be critical.

5 Related Work

Predicting job failures in High-Performance Computing (HPC) systems has garnered significant attention due to its potential to enhance resource utilization and system efficiency. Several studies have explored various methodologies to address this challenge.

Machine Learning Approaches: Antici et al. (2023) proposed an online job failure prediction model that combines classical machine learning algorithms with Natural Language Processing (NLP) techniques. Their approach, evaluated on a dataset from the CINECA HPC center, demonstrated promising results in real-time job failure prediction [1]. Similarly, Banjongkan et al. (2021) developed predictive models using decision tree algorithms (C5.0, CART, and CHAID) to forecast job failures at both the job submit-state and

start-state. Their models achieved high accuracy, with the C5.0 algorithm performing best, and highlighted the importance of job state timing in prediction effectiveness [2].

Feature Engineering and Model Evaluation: Jauk et al. (2019) conducted a comprehensive survey on fault/failure prediction in HPC systems, emphasizing the need for consistent terminology, standardized datasets, and appropriate evaluation metrics. They noted that traditional machine learning metrics might not be sufficient for production HPC environments, advocating for more robust evaluation frameworks [7].

Resource Prediction and Optimization: Mohammed et al. (2019) explored failure prediction in virtualized HPC systems using machine learning techniques. Their study focused on predicting failures based on system logs and highlighted the challenges of handling large-scale distributed data in cloud-based HPC environments [8].

These studies underscore the importance of integrating machine learning techniques with domain-specific knowledge to predict job failures effectively in HPC systems. Our work builds upon these foundations by leveraging PyTorch Tabular for model training and exploring the impact of distributed training environments on model performance.

6 Conclusion and Future Work

In this work, we developed an end-to-end pipeline for predicting job failures in a HPC environment by integrating large-scale log processing with distributed machine learning.

Our performance analysis demonstrated significant speed-up in Spark query execution with increasing cluster size up to 32 nodes, while highlighting the diminishing returns and overheads beyond this scale. In the PyTorch stage, distributed training using the *Distributed Data Parallel* strategy showed promising convergence behavior, albeit constrained by limited computational resources and data.

For future work, several avenues remain to be explored to improve both data processing and model training phases. First, optimizing data partitioning and exploring alternative input formats could further enhance Spark performance and cluster utilization. Second, comprehensive hyperparameter tuning and implementation of early stopping mechanisms in the distributed training framework could yield better model convergence and efficiency. Additionally, experimenting with different optimizers and learning rate schedulers may improve training stability in large-scale settings.

Finally, extending the pipeline to incorporate real-time log streaming and failure prediction could provide operational benefits in HPC resource management. Incorporating more sophisticated feature engineering techniques and exploring ensemble or hybrid modeling approaches may also improve prediction accuracy. Overall, this work lays a foundation for scalable, interpretable, and efficient job failure prediction in HPC systems, with many promising opportunities for continued research and development.

References

[1] Francesco Antici, Andrea Borghesi, and Zeynep Kiziltan. 2023. Online Job Failure Prediction in an HPC System. *arXiv preprint arXiv:2308.15481* (aug 2023). doi:10.48550/arXiv.2308.15481

[2] Anupong Banjongkan, Watthana Pongsena, Nittaya Kerdprasop, and Kittisak Kerdprasop. 2021. A Study of Job Failure Prediction at Job Submit-State and Job Start-State in High-Performance Computing System: Using Decision Tree Algorithms. *Journal of Advances in Information Technology* 12 (01 2021), 84–92. doi:10.12720/jait.12.2.84-92

[3] DevGenius Blog. 2025. From Out Of Memory To Optimized: Handling Java Heap Space & GC Overhead Limit Exceeded Issues In Spark. https://blog.devgenius.io/from-out-of-memory-to-optimized-handling-java-heap-space-gc-overhead-limit-exceeded-issues-in-61f35bb253da. Accessed: 2025-05-01.

[4] Rakesh Chanda. 2025. Spark Out Of Memory Issue — Memory Tuning And Management In PySpark. https://medium.com/@rakeshchanda/spark-out-of-memory-issue-memory-tuning-and-management-in-pyspark-802b757b562f. Accessed: 2025-05-01.

[5] DeepSpeed Team. 2024. DeepSpeed: Communication Efficiency. https://www.deepspeed.ai/training/#communication-efficiency. Accessed: 2025-06-02.

[6] Sumanth R. Hegde. 2024. Everything about Distributed Training and Efficient Finetuning. https://sumanthrh.com/post/distributed-and-efficient-finetuning/ Accessed: 2025-06-02.

[7] David Jauk, Dai Yang, and Martin Schulz. 2019. Predicting Faults in High Performance Computing Systems: An In-Depth Survey of the State-of-the-Practice. In *SC19: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. doi:10.1145/3295500.3356185

[8] Bashir Mohammed, Irfan Awan, Hassan Ugail, and Muhammad Younas. 2019. Failure prediction using machine learning in a virtualised HPC system and application. *Cluster Computing* 22 (06 2019). doi:10.1007/s10586-019-02917-1

[9] PyTorch Tabular Documentation. 2025. Core Classes API Documentation. https://pytorch-tabular.readthedocs.io/en/latest/apidocs_coreclasses/. Accessed: 2025-05-29.

Received 02 June 2025