UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITY OF MODENA AND REGGIO EMILIA

"Enzo Ferrari" Department of Engineering

Master's Degree in Artificial Intelligence Engineering (LM-32)

Queue-State Learning:
Job Wait Time Prediction in HPC Job Queues

Supervisor:
Prof. Lorenzo Baraldi

Candidate:
Davide Agostini
Student ID 196570

ACADEMIC YEAR 2025/2026

Abstract

Queue-State Learning: Job Wait Time Prediction in HPC Job Queues

The significant focus on artificial intelligence over last years has led to growing interest in High-Performance Computing (HPC), particularly in the areas of parallel computing, resource allocation and job scheduling. Among the many systems available, SLURM has established itself as a leading player in the areas of job allocation and shared resource usage within clusters. Among the various features that have enabled it to establish itself in its field are: extremely high scalability; open source; large community support; extremely flexible policies; and excellent integration with the most modern computing systems. Despite the utmost importance of these systems in the modern technological landscape, the optimisation of various parameters is mainly carried out through empirical methods and arbitrary choices, dictated by common sense, made by system administrators. This is precisely the context in which this work fits in, aiming mainly to study the simulation possibilities of a SLURM system through machine learning models. The research focused on developing a predictive model capable of estimating work queue waiting times. In particular, by exploiting Transformer-based architectures, it was demonstrated that it is possible to model dependencies in job submission and scheduling behaviour. Using Slurm logs, the model processes a representation of the internal state of the cluster in order to predict queue waiting times. This system opens the door to possible models for speeding up simulations and, with further research, adaptive implementations for optimising cluster workloads.

Keywords: Transformer, Scheduling, Regression, SLURM, cluster optimization.

Contents

List of Figures

List of Code

1. Introduction

This thesis will describe all the approaches that were followed in the initial study about automation of the SLURM scheduling system. It will describe the analyses performed on the AImageLab SLURM log, and the prediction pipeline to prove the capacity of transformer to understand the underlying logic of slurm scheduling system. The work will be divided in: dataset construction, model algorithm, training, and the validation analyses.

1.1 Motivations

The growing spread of computationally intensive applications, both in industry and science, has led to a proliferation of systems designed to optimise the sharing of computing resources [44]. In particular, this optimisation has been driven by the ever-increasing demands of the machine learning fields, mainly focused on GPU-based HPC systems.

Among the various systems, one of the most known and capable established itself in the sectorwas SLURM (Simple Linux Utility for Resource Management) [26], one of the most famous schedulers currently on the market. Among the various features that have led it to establish itself as a leader in the current market are: open sourceness, a feature that has favored its adoption by universities and public bodies; a scalable and modular architecture, a key element in enabling it to adapt in various context ranging from small to large systems; and an extremely flexible planning policies, capable of adapting well to various operational processes. All these characteristics have allowed the creation of an extremely present community, consolidating its position and leading it to be the most present system in the TOP500 ranking. All these advantages, however, come at a cost in system optimization: the high flexibility forces the various system administrators to select the numerous parameters as carefully as possible in order to maximize system utilization.

Despite the imperative importance of optimizing these systems, to date, most research has focused on simulation [52] or simple analysis methods regarding the history of workloads, leaving more intelligent machine learning-based optimization systems as a niche yet to be explored in depth.

It is precisely in this niche that we want to focus on in this work: analising on the capabilities of

modern transformer systems to represent adequately an HPC system. To demonstrate this, various, fine-tuned, LLMs will be used (e.g. Llama3 [17], QWEN [5]) to predict the queue waiting time of the various jobs, so as to demonstrate their capabilities not only in managing the deterministic elements of the (e.g. priority calculation) but also all those stochastic elements typical of the HPC environment (e.g. crashed jobs).

1.2 Content List

A brief summary of all the contents that will be presented in the next chapters:

- **Background** It will present in depth all the most salient points regarding SLURM system scheduling, provide a description of the main architectures used during the thesis work and the main optimization methods used during trining.
- **Model and Dataset** The composition of the dataset with related analyses, the various components of the model, the inference algorithm and the training methodology will be described here.
- **Results and validation** Here will be analised the obtained results.
- **Conclusions and Future Works** where will be summarized the work and described possible development of this work.

2. Background

2.1 High Performance Computing

As the complexity of the most modern tasks grows, High Performance Computing (HPC) is born, that is the entire set of hardware and technical infrastructures linked to maximizing computing capabilities. HPC systems arise from the combination of highly parallelizable architectures and specific programming paradigms to make the most of these hardware systems; applying these principles therefore makes it possible to manage highly complex operations or large data moles. Numerous are the application areas: climate modeling; simulation of complex systems; bioinformatics; computational chemistry; analysis of large datasets and the development of complex ML systems, first and foremost Large Language Models (LLMs).

2.1.1 HPC Infrastructures

Modern HPC systems consist of clusters of nodes connected by high-speed, low-latency networks; these typically include one or more multicore processors, hardware accellerators, and memory hierarchies designed to maximize data transfer efficiency. The typical scheme of an HPC cluster can be described as follows: **Computational Node**, **High-performance interconnection networks** such as Infiniband [7], **A general-purpose storage solution** used to store applications and user data. This basic structure is complemented by numerous other systems such as distributed file systems; workload allocation schedulers; identity management for managing user access within the cluster; and various others.

2.1.2 Parallelism

The core principle of HPC is parallelism, the reduction of a task into subtasks that can be performed simultaneously on multiple computing units. There are two main forms that parallelism can take:

- **Task Parallelism**, where distinct, independent tasks or are executed simultaneously across multiple processors.

- **Data Parallelism**, where data is separated into smaller chunks to which the same operations are applied.

To fully exploit these forms, various parallel programming models have emerged. Among the most famous there are OpenMP, used primarily in multi-core systems, and CUDA, developed by NVIDIA to make the most of modern GPUs.
All of this was born to solve the main challenge of the HPC world: scalability, "the ability of a (software or hardware) system to handle a growing amount of work efficiently" [6]. Two are the main notions of scalability:

- **Strong Scalability**, which describes the improvement of a fixed size problem as more processes are added. This is described by *Amdahl's Law* [3]:

$$S = \frac{1}{s + \frac{p}{N}} \tag{2.1}$$

  That shows how the maximum theoretical speedup is limited by the non-parallelizable component of the system.

- **Weak Scalability** describing the improvement of problem which size increase proportionally to the number of processes. This is described by *Gustafson's law* [18]:

$$S = s + p \times N = N - (N - 1) \times s \tag{2.2}$$

Current research focuses on novel hardware architectures, scalable algorithms, and optimization techniques that simultaneously improve performance and energy efficiency.

2.2 Simple Linux Utility for Resource Management

In HPC systems, to ensure high scalability and optimal infrastructure utilization, it is essential to have systems that allow for efficient resource management; it is therefore imperative to have systems that are able to allocate the various resources (CPU, memory, and hardware accelerators) according to specific policies defined by the administrator. These systems are known as Job Schedulers One of the most popular cluster management systems in the HPC context is Slurm (Simple Linux Utility for

Resource Management), an open-source workload manager designed for Linux clusters, currently, very popular in academia and the the most popular system within the TOP500 supercomputers. The services made available include user planning, monitoring and control of various jobs; advanced resource allocation mechanisms and high freedom in queue management by the administrator [26].

2.2.1 SLURM Architecture

There are three key functions of the SLURM system: assignment, exclusive or not, of resources to users for specific periods of time, provides a famework for starting, running, and monitoring jobs on specific nodes, resource conflict resolution through the use of a queue system for pending jobs.

To ensure full functionality, the system is essentially divided into two main components: *slurmd*, a daemon running on each computational node; *slurmctld*, a central daemon running on the management node; to interact with these two systems, some command-line directives are exposed as shown in figure 2.1.

As for *slurmd* (SLURM local deamon), it's a multi-threaded daemon running on every node; its main functions are: communicating the state of the machine and jobs to the controller node; starting, monitoring, cleaning up after running the various processes, all under the command of slurmctld; and Stream Copy Service and job control, allowing asynchronous interactions through signal propagation.

Slurmctld is the controller, it must not be run with root privileges and consists of three main components: the *Node Manager*, which is responsible for monitoring the state of the cluster nodes, periodically communicating with the slurmd daemons ensures that the various nodes are in the desired configuration before being considered available; *Partition Manager*, a node grouping system so that specific access rules or job limits can be applied to the various groupings; and, finally, the *Job Manager*, called periodically (e.g. when a node changes state), takes care of handling requests from users, placing pending jobs within the queue, and contacting slurmd to allocate jobs that, based on priority, can be started.

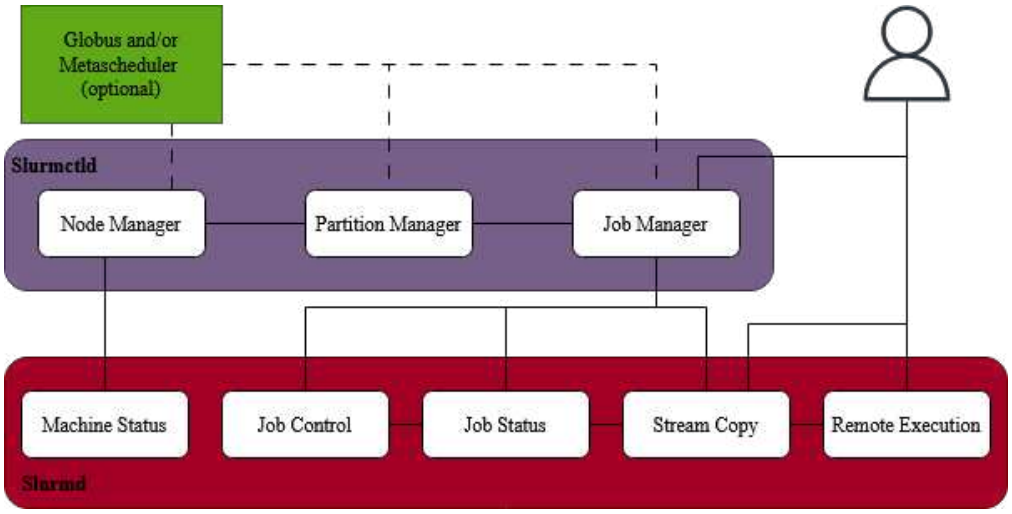


Figure 2.1: Slurm architecture [26]

2.2.2 SLURM scheduling

The scheduling system is the one by which is determined the order of execution of the pending jobs. There are two scheduling routines: a shorter and more frequent one, which occurs whenever it is triggered by events such as a job submission, which takes into account a limited number of jobs (default_queue_depth); and a second, the less frequent and complete one, which takes care of reordering all jobs while respecting the partition_job_depth parameter.

In both cases the jobs are evaluated in priority order, which is calculated taking in account:

- Age, the time waited inside the queue, it increses based on how long it waits inside the queue.
- Association, a factor setted by the admin relative to the association.
- Fair-share, a value that represent the portion of resources promised and consumed; it promotes a fair use of the system to those who have used few resources.
- Job Size, which depends on the number of CPUs and nodes allocated. Its behavior can change by changing the PriorityFavorSmall parameter, in that way is possible prioritizing small or large jobs.
- Nice, a factor controlled by the user to prioritise certain jobs.
- Partition, a factor associated with the node partition.

- QOS, depending on the Quality Of Service.
- Site, a factor added by the admin or by the site_factor plugin. It is the factor that allows admins to inject their own priority calculation.
- TRES, a factor depending on which and how many trackable resorces (GPU, tmpfs) have been requested.

A specific weight can be assigned to each of the above factor, allowing system admin to have fully control on the priority calculation. All factors are expressed as a floating point beloging to $[0.0, 1.0]$ while the weights are expressed as 32 bit integer in the range $[0, 4294967295]$. In addition to simple priority-based execution, smarter systems are needed, especially for large clusters; to address this need, SLURM implements backfilling and preemption.

Backfilling is a technique for improving cluster efficiency, focusing on reducing resource down-time. Considering a typical scenario: we consider a job J_a, with priority value P_a requiring a large number of resources, to be in a PENDING state due to partial resource availability; such resources would be sufficient to complete a J_b job with priority P_b such that $P_b < P_a$. In case of simple scheduling, based only on priorities, these available resources would remain pending until the J_a start-up. Through the backfill technique, however, if the execution of J_b does not delay the execution of J_a, J_b is executed earlier to improve the resources usage.

Preemption, instead, refers to the possibility, for higher-priority jobs, to interrupt lower-priority ones that are currently running. It is implemented as a variation of Gang Scheduling, a system whereby multiple jobs are allocated to the same resources and periodically suspended and restarted. The preempt modes are:

- **OFF** to disable preemption.
- **CANCEL** in this case the job is cancelled.
- **GANG** enables the Gang Scheduling.
- **REQUEUE** requeue the jobs if possible, if not the behaviour is identical to CANCEL.
- **SUSPEND** the job will be suspended until it's possible to resume it. It uses GANG scheduler so GANG option must be setted on cluster level.

2.2.3 Fairshare

Fairshare is one of the main factor in **multifactor priority system**. The *fairshare* component ensures that all the users who have consumed a large portion of available resources receive lower priority for the jobs. There are two main system to calculate this factor: the first and older one is the *Classic*; the newer and more fair, default from SLURM 19.05, is *Fair Tree* algorithm.

The **Classic** algorithm [48] was the primary algorithm until version 19.05. Slurm accounting allows any account to allocate resources to sub-accounts, so a recursive calculation is required to compute this value accurately. To better capture resource usage, which is not constant over time, a weighting factor related to time intervals is applied in the calculation. The less time that has passed since resource consumption, the greater its effect on the calculation of the fairshare parameter.

The **Fair Tree** algorithm [49] is a newer version, developed to solve some flaws in the older algorithm. In fact, as shown in [47], in various situations the classic Fairshare algorithm was not fair at all, giving higher priority to accounts with higher resource usage. The main advantage of this algorithm is the hierarchy of fairness: if an account A has a higher fairshare factor than one of its siblings, all of A's sub-accounts will have a higher fairshare factor than the others.

2.3 Regression

Regression is one of the various tasks in the supervised learning field; it is a core component of various ML pipelines across multiple fields, such as economics, finance, biology, and so on. In the regression task, we aim to obtain the relationship between the input and the output space. This relation is represented as a function f defined as

$$f : X \to Y$$

where X is known as the independent variable, while Y is the dependent variable [54].

Given a supervised dataset $\mathcal{D} = (x_i, y_i)_{i=1}^n$, with $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, a regression model aims to learn a function $f : \mathbb{R}^d \to \mathbb{R}$ that approximates the dependence of Y on the features X, minimizing an error criterion and, if desired, quantifying the uncertainty associated with the predictions. This approach is common both in classical statistical analysis and in the learning-oriented formulation used in AI.

The modelling choice depends strongly on the expected relation $X \mapsto Y$, dimensions, presence of non-linearity, interpretability, robustness, outlier presence.

2.3.1 Regression Models

The simpler model is the **Ordinary Least Squares** (OLS) and serves as a valid loss function as long as six properties are valid: *Linearity, Constant Error Variance, Indipendent Errors, Multicollinearity Absence, Data Normality, No Exogenity* [54]. Under these assumption OLS is a valid function and, if $J(\theta)$ is convex the solution is closed.

$$J(\theta) = \sum_{i}^{n} (y_i - (\theta x_i + b))^2 \tag{2.3}$$

$$\theta^\star = (X^\top X)^{-1} X^\top Y \tag{2.4}$$

Polynomial Regression is the generalization of the of linear regression. Instead of considering only linear function we try to approximate a function defined as

$$y_i = \sum_{n=0}^{N} \beta_n x_i^n \tag{2.5}$$

Some strategies, like **Lasso** [55] or **Ridge** apply some regularization to the parameters.

For more general approximation functions there are various possibility:Support Vector Regression with Kernel trick and regularization terms; Regression tree and Tree ensemble (like Random Forest or Gradient Boosting) and General Neural network.

2.3.2 Metrics

Evaluating a regressor require two element: goodness of fit in training and generalization ability to unseen data. Standard metrics include:

$$\text{MSE} = \frac{1}{n} \sum_{i} (y_i - \hat{y}_i)^2 \tag{2.6}$$

$$\text{RMSE} = \sqrt{\text{MSE}} \tag{2.7}$$

$$\text{MAE} = \frac{1}{n} \sum_{i} |y_i - \hat{y}_i| \tag{2.8}$$

MSE/RMSE penalizes large errors more severely while MAE is more robust to outliers.

The coefficient of determination R^2 measures the share of variance explained with respect to a null model (intercepts only); R^2 "adjusted" corrects (in part) optimism as d, which is the input dimension, increases. Usually is common practice to combine both error and R^2 metrics.

2.3.3 Training Loss

The most widely used regression loss is the MSE, which popularity derives from the mathematical simplicity, differentiability and the quadratic nature, which leads to a strong penalization for large errors. However, because squaring amplifies extreme residuals, MSE is sensitive to outliers and may lead to unstable estimates when the dataset contains anomalous observations. An alternative is MAE, which is more robust to outliers, unfortunaltely the optimization in this case is less smooth.

Between these two extremes, the Huber loss [15] offers a great compromise: it behaves quadratically for small residuals and linearly for large ones, combining the stability of MSE with the robustness of MAE. His definition is

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{if} \quad |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{2.9}$$

Another famouse loss is the Quantile Loss [29] that, instead of predicting the conditional mean, like MSE, predicts a conditional quantile (depending on the parameter $\tau \in [0, 1]$)

$$L_\tau(y, \hat{y}) = \max(\tau(y - \hat{y}), (1 - \tau)(\hat{y} - y)) \tag{2.10}$$

In that way there's an asymmetric relevance if the error is caused by an undervaluation or an overvaluation.

2.4 Neural Network

Artificial neural networks are the core element of deep learning. They are biologically inspired computational models based on the idea that a large collection of simple elements can lead to complex behavior, in particular by approximating a general function $f : \mathbb{X} \mapsto \mathbb{Y}$.

The main inspiration is the human neuron [35]. To build a network, various ones are connected and organized into layers. The neuron structure is extremely simple; it is based on two main

elements: a linear function and a non-linear activation function. Each neuron receives the input values from the previous layer, applies a linear function (a weighted sum plus a bias term), and then applies a non-linear function. This is the neurons output. In a more formal notation, the output y is defined as

$$y = \phi(w^\top x + b) \tag{2.11}$$

The function $\phi(\cdot)$ represents the nonlinear activation like sigmoid, hyperbolic tangent, or rectified linear unit (ReLU); this is a fundamental component, in fact, since the each layer can be written in matrix form

$$\mathbf{h}^{(l)} = \phi\Big(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\Big)$$

where $\mathbf{h}^{(l)}$ is the **l**-th layer state,$\mathbf{W}^{(l)}$ weights matrix, and $\mathbf{b}^{(l)}$ the bias vector, knowing matrix multiplication properties is enough to undertand that non-linearity absence would lead to collapse in a simple linear function.

The standard feedforward network, also known as **multilayer perceptron** [45], shown in figure 2.2, is organised in one input layer, one or more hidden layers, and an output layer. The input layer receives the observed features; the hidden layer transforms these features into more refined internal representation, and the output layer produces the final prediction. In the standard architecture, all the neurons of a layer view all the precedent ones, that leads to an exponential growth of the parameters, which is one of the main drawbacks of the architecture.

Another great flaw of these systems is the lack of interpretability. In fact, even if the neural component is extremely simple and, with minimal mathematical knowledge, is completely understandable, the interaction of multiple neurons quickly removes any sort of interpretability, resulting in a predictor which is basically a black-box model.

Despite these drawbacks, the neural networks have been the lead technology in the field for numerous years, mainly for their learning capability. In fact, as shown in [22], neural networks, with few hidden layers, can approximate any continuous function to any degree of accuracy. This capacity makes them one of the most powerful systems available today, despite their computational cost.

To adapt the system to various domains, a lot of other interesting solutions have been studied: for example, in the Computer Vision domain, one of the most studied architectures was the

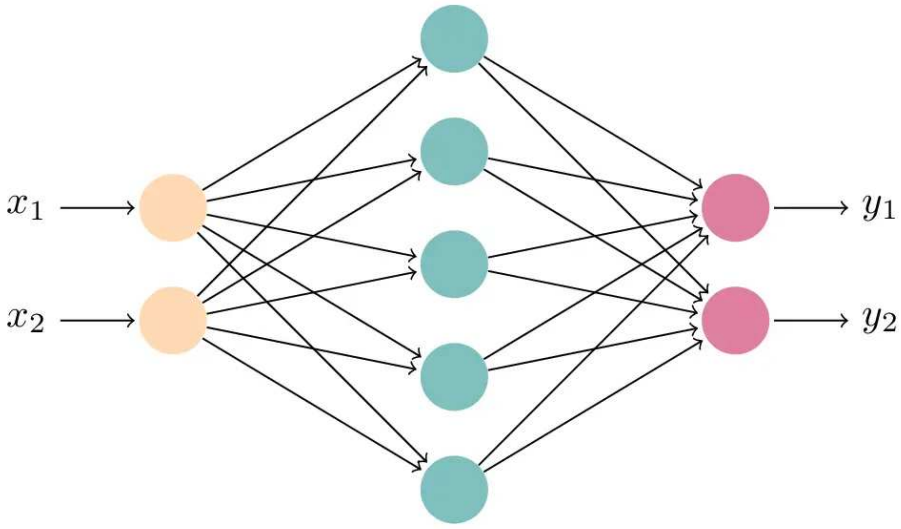


Figure 2.2: The multi-layer perceptron scheme.

Convolutional Neural Network (CNN) [30] [19] or, for sequence analysis, the *Recurrent Neural Network* [14] [21].

2.4.1 Training

The training is the iterative process through which we update the parameters of a Neural Network; having train-set data and a loss function we try to reduce the prediction error and improve generalization on unseen test examples.

Training is, basically, an optimization problem consisting in finding the parameters $\theta^\star$ that minimise the loss function over a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$. Considering $f(x;\theta)$ the neural network function, the most general version of the empirical risk that we want minimise is

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\big(f(x_i;\theta), y_i\big) + \lambda\,\Omega(\theta) \tag{2.12}$$

where $\mathcal{L}(\cdot)$ is the task-specific loss and $\Omega(\theta)$ a regularization term, like an l_2 penalty. The method, generally, used to find the minimum is **Gradient Descent**, an iterative optimization algorithm that updates the parameter in the $\nabla_\theta J(\theta_t)$ opposite direction [8]. The gradient calculation is computed efficiently by backpropagation, that applies gradient chain rule through all the network's layers [46]. The basic step is defines as:

$$\theta_{t+1} = \theta_t - \eta_t \nabla J(\theta_t) \tag{2.13}$$

where $\eta_t > 0$ is the learning rate. Unfortunately for large neural networks and large datasets the computation of the exact gradient is too expensive so, instead of using the full set, the optimisation is performed using mini-batch randomly sampled, this method is called **stochastic gradient descent** (SGD). Despite is utility this stochastic method add some noise into the optimisation process and so, to reduce it, various solution have been studied: adding momentum to SGD, Adam [28], Adagrad [13] and many others. Despite the great adavantage for reaching fastest convergence, mantaining statistics of past gradients represents a substantial memory overhead. We will get back to that argument in secion 2.11

2.4.2 AdamW

AdamW is one of the most famous variants of Adam, the main reason is that his variation is simple but very powerful: decoupling the weight decay from the gradient update [34]. Standard Adam maintains exponential moving averages (EMA) of the first and second moments of the gradient:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Where g_t is the gradient at the step t, β_1, β_2 are two constant in range $(0, 1)$.

The use of the momentum makes the trajectory of the optimization more smooth and allow the algorithm to converge faster, while second moment, instead, is used to adapt the learning rate. The final update will be[1]:

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{2.14}$$

The important, yet effective, change brought by AdamW regards weight decay. In older implementations, regularization was applied directly to the gradient but for adaptive optimizers, like Adam, that was not equivalent to a weight decay on the parameter update. To solve that flaw the authors [34] simply move the normalisation into the weights update:

$$\theta_{t+1} = \theta_t - \eta_t \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} + \lambda \theta_t \right) \tag{2.15}$$

[1]The terms $\hat{m}_t$ and $\hat{v}_t$ are a regularised version since, at the beginning are biased towards zero.

This simple change strongly stabilise the training and, empirically proven by [34], leads to a better generalisation performance.

2.5 Transformer

Transformer networks [56] are one of the greatest expressions of machine learning today, created for Natural Language Processing are today implemented in the majority of fields thanks to their great versatility. Before that, for sequence processing, SOTA architecture there were Recurrent Neural Networks [14], among which Long Short-Term Memory (LSTM) [21] and derivations stood out. What made these systems inefficient was the sequentiality that, during training, made parallelization excessively complex. Transformer architecture, on the other hand, by exploiting the attention mechanism, allows us to bypass the sequentiality problem and introduce a fully parallelised training.

As regards Figure 2.3, the model was born as a translation model. It takes as input a sequence $X = [x_0, x_1, ..., x_n]$ in the component called encoder, and generates a sequence $Y = [y_0, y_1, ..., y_m]$ from the component called decoder.

2.5.1 Attention Mechanism

The attentional mechanism represents the main element of the transformer architecture. Its main feature is the capability, for each sequence element, to observe each other element of the sequence, the number of viewable elements is called the *receptive field* and, in the case of the transformer, this quantity is theoretically infinite. The attentional operator, to be applied, requires three distinct sequences of elements: queries, keys and values.

$$Q \in \mathbb{R}^{n_q \times d_k}, K \in \mathbb{R}^{n_k \times d_k}, V \in \mathbb{R}^{n_k \times d_v}$$

Given these Q, K, V the usual attention formula is:

$$\text{attention}(Q, K, V) = \text{softmax}(\frac{QK^\intercal}{\sqrt{d_k}})V \tag{2.16}$$

This dot product is the classical definition of attention.To preserve the variance the normalization term $1/\sqrt{d^k}$ is applied. Usua lly the dimensions d_k and d_v are the same, this allows for easier

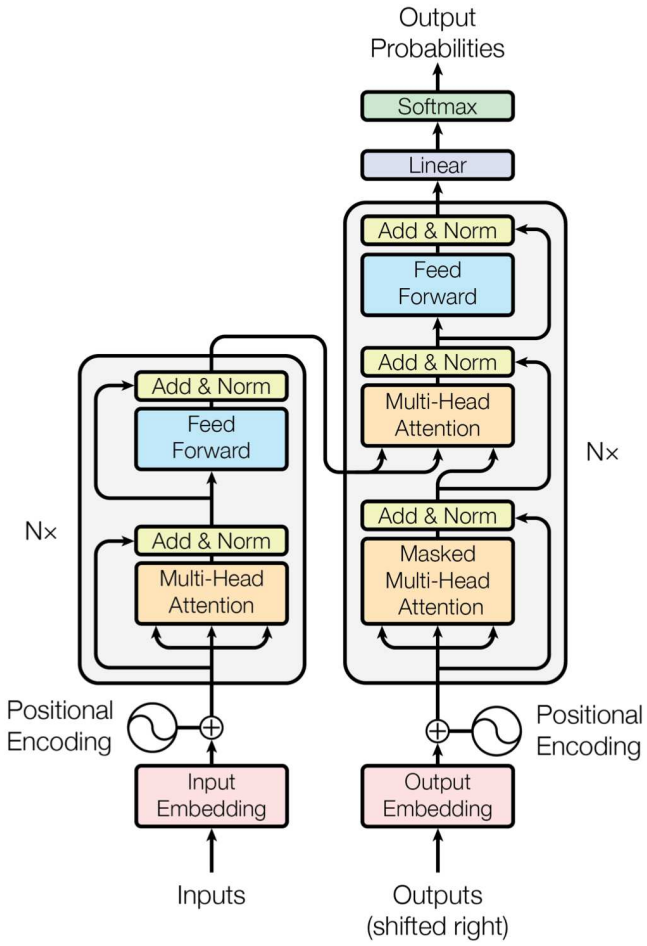


Figure 2.3: Transformer encoder-decoder architecture [56]

management of the architecture and, furthermore, the possibility of applying residual connections, mainly to avoid phenomena such as gradient exploding or gradient vanishing [19]. After the operation the output will be:

$$\text{output} \in \mathbb{R}^{n_q \times d_v},$$

So, in the output, we will always keep the same number of element the we have fed in input. The final output, since the *softmax* transforms the inputs in probability scores, will be the weighted mean of the values vectors. The attention score (weights) will depends, instead, on the similarity between the keys and the queries.

In 2.3 there are 2 implementations of attention

- In the encoder the implementation is called **Self Attention**, the three sequences Q, K, V are all generated from the same input sequence $X = [x_0, x_1, ..., x_n], x_i \in \mathbb{R}^d$ through a learned

linear projection.

$$Q = XW^Q, K = XW^K, V = XW^V$$

- In the decoder the second attention is called **Cross Attention**, that is the communication channel from encoder to decoder. In this case Q, K, V do not come from the same sequence but K, V both come from the encoder while Q arrives from the decoder.

2.5.2 Masked Attention

In various real systems causal dependencies exist within sequences, and these cannot be understood before the conclusion of the sequence itself. During training, in these scenarios, it is therefore necessary to limit the view of each element to only specific other ones. This process is called Masked Attention. Its application acts directly on the similarity matrix, replacing all masked values with an extremely negative value, thus setting that value to zero once softmax is applied. There are various types of masking, in the classical transformer, during training, is applied the "causal" or "triangular" one 2.4.

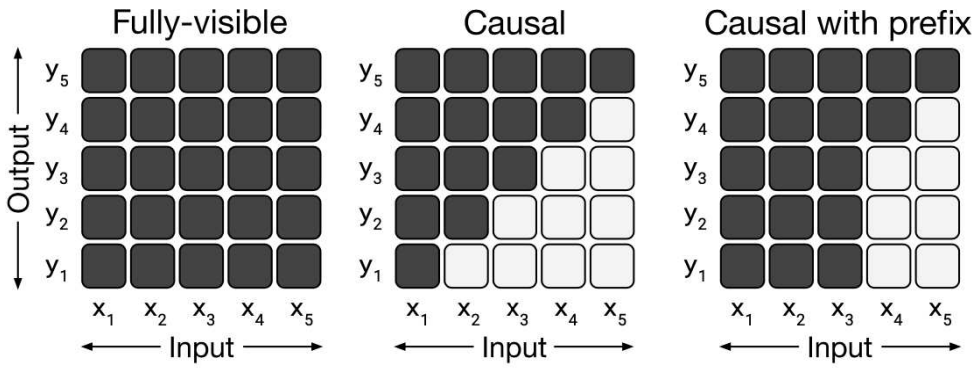


Figure 2.4: Various type of masking

2.5.3 Multi-Head Attention

As said previously, the projections that are applied to the sequences are among the various parameters that are learned during training. To improve the transformer's learning capabilities, and make the most of the parallelization system, it's possible to compute various projections in each layer and,

after it, apply various separate attentions. This system is called multi-head attention. The usual implementation is:

$$\begin{aligned} \text{MultiheadAttn}(X) &= \text{Concat}(\text{head}_1, ..., \text{head}_n)W^O \\ \text{head} &= \text{Attention}(XW^Q, XW^K, XW^V) \end{aligned} \tag{2.17}$$

2.5.4 Structure

The left part of figure 2.3 is called the *Encoder*, it consists in a sequence of identical layers composed by: A **Multi-head self-attention**; a **LayerNorm** [4], a normalization layer that apply, over the mini-batch, the function:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}(x) + \epsilon}} * \gamma + \beta \tag{2.18}$$

with γ and β learnable parameters; a **Position-wise Feed Forward Network**, which is a Feed Forward network applied to each element of the sequence indipendently.

$$FFN(X) = \max(0, xW_1 + b_1)W_2 + b_2$$

and, lastly another **LayerNorm**. As said above, since we usually keep the same dimension for the queries and the values spaces, we can add a residual connection [19] to stabilise the training both after the Attention and the Feed Forward.

The right part is called *Decoder*, is similar to the encoder but introduces an additional attention layer. Each decoder layer consists in the next sub-components: A **Masked Multi-head self-attention**, folowed by a **Cross-attention** to inject information from the encoder to the decoder and, lastly a **Position-wise Feed Forward Network**. Also in this component, after every sublayer, a **LayerNorm** and a residual connection are applied.

2.5.5 Positional Encoding

Considering how Q, K, V are generated in self-attention and how attention is performed from equation 2.16 it's easy to observe that the attention operator is position invariant. Since the first application of [56] is in NLP such a property is completely unacceptable. To inject some positional information at the input is added a, so-called, *positional encoding*: a vector of the same

dimensionality of each input element that embeds positional information. The original encoding in [56] was generated with waveforms:

$$PE(pos, i) = \begin{cases} \sin(pos/10000^{\frac{2i}{d}}) \text{if} \quad i \mod 2 = 0 \\ \cos(pos/10000^{\frac{2i}{d}}) \text{if} \quad i \mod 2 = 1 \end{cases} \tag{2.19}$$

This choice has been made, primarly, to help the model to express relative position, in fact any postion $pos + k$ can be represented as a linear function of pos. The other reason, in contrast with the other main system at the time (*learned positional embedding*) is the capability to generalize sequences longer than the ones viewed during training. Other very famous solution for positional information are **RoPE** [53] and **ALiBi** [41].

2.5.6 Inference

In [56], as said before a model for NLP, the model generate an output sequence $y_1, y_2, \ldots, y_T$ starting from an input sequence $x_1, x_2, \ldots, x_S$, in this case for machine translation.

The elements of the sequence are not words or single letters but tokens, smaller units of text that can be words, endings, single letters. The operation to split a sentence is tokens is called **Tokenization** and is necessary for the construction of the sequence. Once the various tokens are generated, they are converted to integer ids and used to retrieve the embeddings of that token from a "dictionary" inside the model. Among these tokens, some are for "special" uses that will be defined later.

Formally the model estimates:

$$P(y_1, \ldots, y_T | x_1, \ldots, x_S) = \prod_{t=1}^{T} P(y_t | y_{<t}, x_{1:S}) \tag{2.20}$$

The probability of each y_t tokens depends on all the x_i tokens and the all the $y_{<t}$, which are the preceding tokens.

As for the encoder part, the system takes as input the entire sequence of tokens X and follows the pipeline described up to now: positional embeddings are added, the sequence passes through all attentional layers and, at the last layer, the refined sequence will be passed to the decoder.

Regarding the decoder, there are some clarifications to be made regarding tokens. The sequence refinement process is identical to that of the encoder except for the use of a cross-attention system,

in this case however the process is generative. The last token in the sequence is used to perform the prediction of the next token, as evident in the figure. At the end of decoder component there is a linear layer and a softmax that transform the results in probabilistic scores, so it is mandatory to have a token from which to start this prediction. This token, given as input to the system, is a special token called a $< bos >$ beginning-of-sequence token. The token predicted by the system will then be added to the sequence Y until an additional special token, end-of-sequence, $< eos >$ is generated that interrupts the gneration process. As obvious the generative process is sequential. The probabilistic output has the same dimensionality of the token dictionary, the highest value is the most probable token.

2.5.7 Training

Speaking about training, the two sequences X and Y are given simultaneously, the way to prevent the previous tokens from paying attention to the subsequent ones, in the decoder part, is the masking system described above. This feature is what has led transformer models to be so diffuse in Language Modelling, the possibility of analyzing sequential systems in parallel during training.

2.6 Mixture of Experts

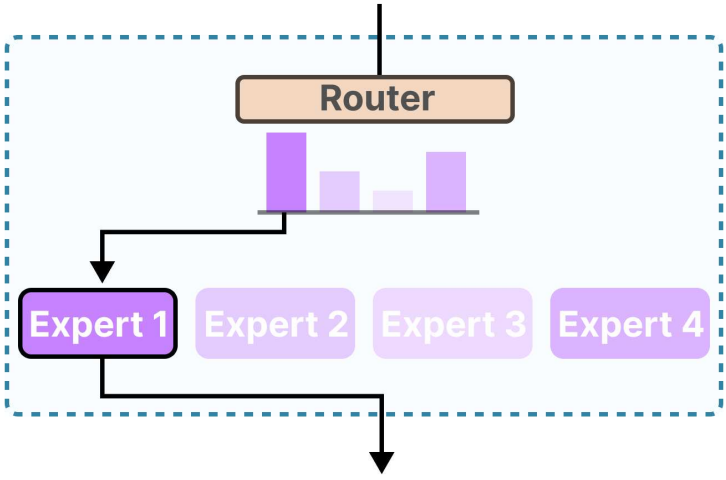


Figure 2.5: MoE block with 1 route

Mixture of Experts (MoE) [25] [51] is a conditional computation architecture primarly designed to increase model capacity without requiring all parameters activation for every input. The main idea is a divide-et-impera strategy: instead of training a monolithic network for solving the entire task, the model is divided in multiple specialized subnetworks, called experts, which, together, with a gating or routing system that decides which experts to send each input to. The renewed interest in MoE is tightly connected to Transformer scaling, in fact sparse gating can massively increase model capacity [51] since, in large language models, the dense feed-forward network (FFN) inside a Transformer block can be replaced by a bank of expert FFNs. The router then chooses which expert processes each token representation.

Formally the MoE can be defined, denoting x as input, by a $E_i(x)$, which is the output of thr i-th expert, and a gating function $G_\sigma(x)$ that outputs also weights relative to the various expert, described as $G(x)_i$. The output formula is:

$$y = \sum_{i=0}^{n} G(x)_i E_i(x) \tag{2.21}$$

In particular, to save some computational capability, when $G(x)_i = 0$ is possible to avoid $E_i(x)$ calculation. A standard gating function can be, for example $G_\sigma(x) = \text{Softmax}(x \cdot W_g)$.

In [51], one of the most famous example of MoE in NLP context, is used this particular gating function:

$$\begin{aligned} H(x)_i &= (x \cdot W_g) + \text{StandardNormal}() \cdot \text{SoftPlus}((x \cdot W_{\text{noise}})_i) \\ \text{KeepTopK}(v, k)_i &= \begin{cases} v_i & \text{if } v_i \text{ is in the top } k \text{ elements of } v, \\ -\infty & \text{Otherwise.} \end{cases} \\ G(x) &= \text{Softmax}(\text{KeepTopK}(H(x), k)) \end{aligned} \tag{2.22}$$

The equation 2.22 is quite self-explanatory; the only unexpected element is the noise addition. That's a strategy for load balancing: during training, without any normalisation, good *experts* tend to receive tokens more frequently; this leads to a feedback effect where strong *experts*, being more chosen, are also often evaluated and, therefore, improved during gradient descent, leading to other performance improvement for that *experts*. So, to avoid *expert* starvation, some noisy output is applied during training.

The advantages of MoE are clear: expert specialization leads different subnetworks modeling different regions or data structures; sparse activation, since only a subset of model receives the

inputs, means that the active computation per example is much smaller than dense feed-forward one and the modular architecture is quite compatible with modern Transformer blocks. At the same time, MoE has persistent weaknesses: routing collapse; as said before, expert starvation, communication-heavy all-to-all dispatch and mainly a train process that, usally, keeps much less stability than dense one. For these MoE should not be viewed as a universal replacement for dense networks, but a mechanism that, thank to exploiting partial conputation thank to its conditional nature, leads to a very goods scaling capacity.

2.7 Bidirectional Encoder Representations from Transformers (BERT)

Bidirectional Encoder Representations from Transformers or BERT is a language representation model that, among other methods, has made pre-training models on large amounts of poorly refined data a standard [12]. In this way you get models with good weights to start from to perform subsequent training (called Fine-Tuning) with labeled data.

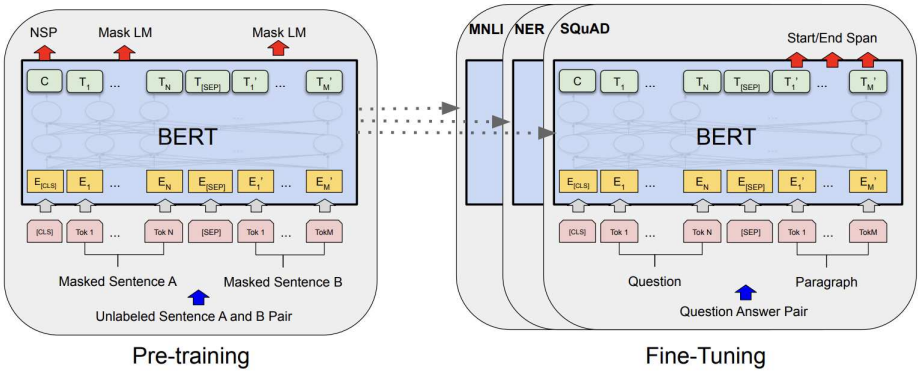


Figure 2.6: Bert pre-training [12]

2.7.1 Motivations

Before Transformer a lot of NLP used static embeddings to represent the "word type" and not the contextual occurrence. Approches like word2vec [38] were extremely scalable but not suit and very limited on contextual variability. To better answer this task multiple approach have emerged,

one of the most famous was ELMo [40] which propose embedding obtained from internal states of bidirectional LSTM. ELMo belogns to the *Feature Based* approach, where model are trained to extract some task specific embedding. On the other hand, model like GPT [42] belong to the *fine-tuning* approach where, after adding few task specific parameters, all parameters are fine tuned in supplementary training steps.

Bert authors main critique was that, specifically in fine-tuning approach, the full potential of transformer architecture could not be achived since the causal masking, avoiding in that way a full understanding of the context in the pre-trained representation. The authors explore a self-supervised pre-training approach composed by two task: **Masked Language Modelling** and **Next Sentence Prediction**.

There have been numerous heirs to this approach, and subsequent literature has reworked these goals: RoBERTa [33] removes NSP and argues that much performance depends on data optimization and scaling; ALBERT [31] proposes SOP (Sentence Order Prediction) as a more consistency-focused task; SpanBERT [27] uses span masking and use the span boundary representation to predict the masked content.

2.7.2 Architecture

The core of BERT architecture is the **Transformer Encoder** explored in section 2.5. Particularly interesting is the addition of two other element that will be used in pre-training tasks:

- As said before, encoder are a permutation invariant architecture so, in NLP, it's mandatory using any kind of positional encoding. To improve the **Next Sentence Prediction** task, to differentiate the two sentences, instead of using a simple positional embedding they have also added a *segment embedding* specific for the first or the second sentence as shown in figure 2.7.

- On top of the token sequence a special [**CLS**] is added. This is a special token used for the classification during the NSP task. Another token, [**SEP**] is used to separate the two sentences. Both embedding are learned at training-time.

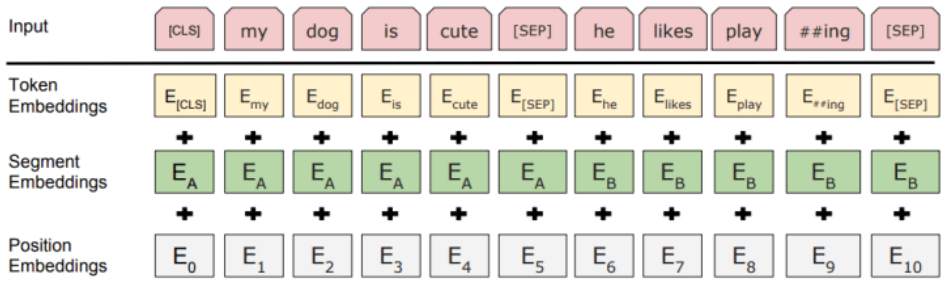


Figure 2.7: Bert Embeddings [12]

2.7.3 Pre-training

The two main tasks, during pre-training are **Masked Language Modeling** and **Next Sentence Prediction**.

About **Masked Language Modeling** the objective of the task is the prediction of masked token. During pre-training, a certain percentage of the tokens are masked (in BERT's case, 15%). After passing through BERT, the masked token is used to predict the correct one. Since the masked token is not present during fine-tuning, to avoid excessive mismatch, the masking strategy is mixed: 80% of the time, the token is substituted with the [**MASK**] token, 10% with a random token, and 10% left unchanged.

For what concerns the **Next Sentence Prediction**, a lot of NLP tasks are based on understanding the relationship between two or more sentences, a skill not included in the task before. To train the model to perform this task, couples of sentences are created; in 50% of the cases, the sentences are consecutive, while in the other half, they are not. As mentioned before, the two sentences are separated by a token and have two different segment embeddings. To predict the sequentiality, the CLS token present at the beginning of the sequence is used.

2.8 Llama

Llama family [17], developed by Meta, has evolved rapidly between 2024 and 2025, moving from mainly textual "generalist" models (Llama 3) to a more articulated "constellation":

- **Llama 3.1** is the big general-purpose text step in the family. Meta introduced 8B, 70B, and 405B versions, with 128K context and broader multilingual support. It is the "flagship"

branch for stronger reasoning, coding, and agent-style use.

- **Llama 3.2** which is the multimodal and edge branch. It implemets small text-only 1B and 3B models mainly for lightweight and on-device uses, plus 11B and 90B vision models for text and images.
- **Llama 3.3** is a more focused, text-only, 70B instruct model, extremely optimised and capable of outperforming Llama 3.1 70B and Llama 3.2 90B. Despite the "low" parameters number, for some specific tasks, it 's able to approach the Llama 3.1 405B performance.

The official Llama 3 models are auto-regressive language models using an optimised decoder-only architecture, while the instruction-tuned variants rely on supervised fine-tuning (SFT) and reinforcement learning from human feedback (RLHF) [39] for alignment specifically each round involve a phase of Instruction Tuning [58] and a second phase of Direct Preference Optimisation (DPO) [43]. One of the main focus of the Llama model is the safety; in fact, in the official Llama 3.1 model card, Meta states that SFT and RLHF have been used to align behavior with helpfulness and safety, targeting mainly risky outputs, refusal behavior, and response tone.

2.8.1 Architecture

The Llama 3 family is built around a decoder-only, autoregressive Transformer architecture with a relatively standard design except for few differences that enhance, mainly, scale and the inference efficiency.

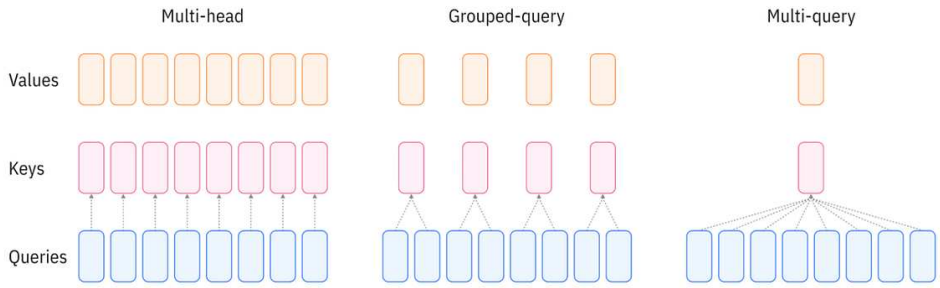


Figure 2.8: Grouped Query Attention [2]

Grouped Query Attention (GQA) [2] is an attention variant that stands in the middle of standard multi-head attention (MHA) and multi-query attention (MQA). In MHA, as said in section

2.5, every attention head has its own query, key, and value projections. In MQA, query heads share just one key and value head, making decoding much faster but hurting prediction performance quality. GQA is the compromise: it divides query heads into groups, and each group shares a single key head and value head.

SwiGLU or Swish-Gated Linear Unit, an activation function that computes two projections of the input and multiplies them elementwise using the SiLU component as a gating function.

$$SwiGLU(x) = Swish_\beta(xW + b) \odot (xV + c)$$

Rotary Position Embedding (RoPE) [53] replaced the absolute positional encoding. RoPE is a positional encoding mechanism designed to easily incorporate both absolute and relative position inside the token. More specifically, the absolute positionin is incorpored by rotating the vector but, as mathematically proven in the paper, when the dot product between a rotated query and a rotated key is computed, the resulting attention score becomes a function not only of content similarity but also of the relative distance between positions. This is one of its main theoretical strengths, since many linguistic dependencies are better characterized in relative rather than absolute terms.

2.9 QWEN

The Qwen family [5] has become one of the most visible open-weight model lines in the contemporary LLM ecosystem. Within that family, Qwen3 [59] and Qwen3.5 mark two distinct but closely related stages of development. Qwen3, released in April 2025, formalized a hybrid reasoning paradigm in which a single model family can operate either in a slower, step-by-step "thinking" mode or in a faster "non-thinking" mode. First released in February 2026, extends this trajectory by moving beyond a primarily text-centered reasoning framework toward a multimodal-agentic approach; In particular, this paradigm shift is most evident in the transition from static language modelling and prompt-and-response interaction to the use of tools, a multimodal approach (image and text-to-text), and agent-based workflows integrated into real-world deployment environments.

2.9.1 Architecture

At the backbone level, Qwen3 shares a lot of similarities with Llama3: both are autoregressive decoder-only Transformer families using Grouped-Query Attention (GQA), SwiGLU feed-forward blocks, and RoPE positional embeddings. For Qwen3, it's explicitly said that its dense models use GQA, SwiGLU, RoPE, and RMSNorm with pre-normalization. From an architecture point of view, the largest difference is this: Llama 3 is fundamentally a dense family, while Qwen3 is a mixed family with both dense and MoE variants, making QWEN family more flexible. Others small differences are reletad to attention geometry, the absence of Q, W, K bias and the presence of a Q, K normalisation to stabilise the training.

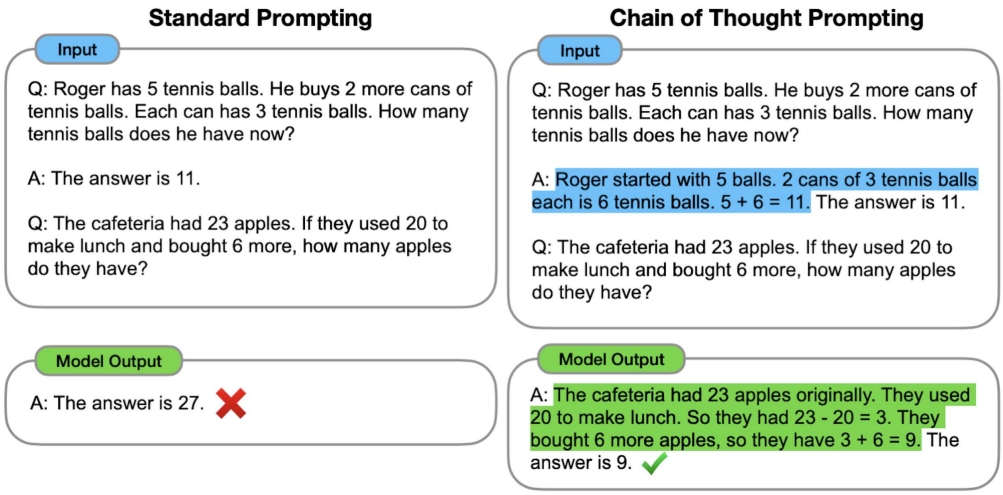


Figure 2.9: Chain-of-Thought from [57]

2.9.2 Thinking mode

One of the main feature of QWEN is the thinking mode. This is an inference function in which the model spends extra computation on explicit multi-step reasoning before producing the final answer, a technique mainly known as **Chain-of-Thought** shown in image 2.9, whereas non-thinking mode is optimized for fast, direct responses. The main advantage of thinking mode is higher reliability on tasks such as mathematics, coding, and complex logical reasoning, where deeper deliberation can improve accuracy; the advantage of non-thinking mode, instead, grants an higher efficiency, lower computation consume a lower latencies for daily chat, where extended reasoning is unnecessary. In

particular, in Qwen3, both behaviors are integrated into the same model family, so users can trade off speed and depth without switching to a separate reasoning model. This integretion is injected in the models during post-training. This process is divided in these phases:

- **Long Chain-of-Thought cold start** is the first phase where highly curated data is used to generate and inject various reasoning patterns without overly emphasizing the immediate performance.
- **Reasoning RL** in which unseen data are used strengthening exploration, planning, and accuracy on hard reasoning tasks. The method applied is GRPO [50].
- **Thinking Mode Fusion** integrates the reasoning behavior with standard instruction-following so the same model can support both "thinking" and "non-thinking" use. One of the main advantage, underlined by [59] is the capability of the model to budgeting the reasoning giving answer that emphasise, for example, the absence of time to reason caused by the wall-time applied by the user.
- **General RL** to improve broader alignment and response quality outside pure reasoning tasks.

This training structure unifies the two modes inside the same model; in particular, the first two tasks are more *thinking*-based, while the others are specifically for *non-thinking* response.

Another stage, non "mode" based is done at the end: for lightweight models is performed a strong-to-weak distillation from larger teacher models instead of running the full four-stage pipeline independently, this choice has been made, primarly, for efficience.

2.10 Parameter Efficient Fine Tuning (PEFT)

The term PEFT refers to all those techniques for performing fine-tuning without having to excessively modify already trained parameters, therefore reducing the consumption of computing resources. A general definition for PEFT could be the adaptation of a pre-trained model f_θ to a specific task learning only a small set of parameters Δ_ϕ (newly added or a subset of the existing one) while keeping most of θ frozen.

Among the various techniques studied there are two families of great importance: *additive*, which are mainly based on adding small layers (LoRA,) and *prompt-based*, which rely on prompt engineering or learning techniques.

2.10.1 Motivations

The classical fine-tuning paradigm, which implies the update of all model parameters, after the diffusion of large models with billions of parameters [17] [59], organizations managing many different tasks in various domains, has fast become completely impractical. The main problem answered by PEFT are: Storage and Deployment, in fact full Finte Tuning requires a full model chekpoint per task PEFT, instead, stores only a subset of the parameters; Training Optimization, since even if activations are required through the backbone, freezing most of the weight substantially reduces optimizer state memory; Operational Flexibility, PEFT, using smaller adapters, allows a fast switches beetween various tasks and, at last, Energy Consumption, since faster and lighter training allows for more efficient and reduced wasting.

2.10.2 Low Rank Adaptation (LoRA)

Low Rank Adaptation (LoRA) [23], in the last years, has become one of the most influential PEFT methods, it adresses the PEFT challenges by freezing the pre-trained backbone weights and learning only a small number of additional parameters. These new parameters are added only to specified layers.

Based on [1], the authors of LoRA theorised that, to update the backbone weights, a full-rank update was not necessary.

Let's consider a general weight matrix $W \in \mathbb{R}^{d\times k}$. Instead of learning a full-rank update ΔW we limit the rank of the update to a certain value. In particular, LoRA, represent the update as the product of two low-rank matrices:

$$\Delta W = BA \tag{2.23}$$

where $A \in \mathbb{R}^{r\times k}$ and $B \in \mathbb{R}^{d\times r}$ with $r \ll \min(d, k)$. The adapted weights are, then, used in parallel

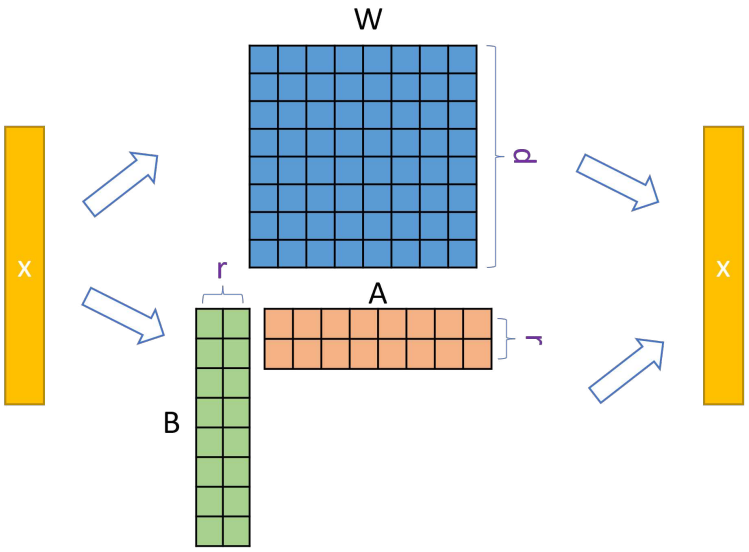


Figure 2.10: LoRA [23] visualization

with the original weights giving, as final result:

$$\begin{aligned} y &= W'x \\ &= (W + \Delta W)x \\ &= (W + BA)x \end{aligned} \tag{2.24}$$

During training only the A and B are trained while the backbone parameter remains frozen. This approach reduces significantly the trainable parameter number, allowing a lower GPUs usage. To stabilize the optimization, a scaling factor is applied to matrix A. Matrix B, instead, is initialised at zero; in that way, the first inference is not influenced by the new modules.

With the new matrix, 2 main approach can be used. If the application, or the domain, is time-sensitive and higher latencies are not acceptable, the matrix ΔW can be simply added to the older one. If, instead, it's necessary to have various adapters for various tasks, it is possible to keep the LoRA module separated and switched time by time for the various tasks.

2.11 Memory Efficient Strategies

With the rapid increase of the model size, finding a clever way to reduce memory footprint has become a very crucial challenge. In this section we will present some of the literature solutions applied to this work.

2.11.1 Quantisation

Among the various strategies, the most famous and most obvious is parameter **quantization**, a process through which the parameters are remapped to a lower numeric representation. Quantisation, in this case, is the mapping of a value set (usually larger and continuous) to another set. One of the main examples is the IEEE 32bit floating point [37]. To perform a general quantization, three steps are required:

- Computation of a normalization constant N and bias b; this moves the values v_i into the range of mapping domain D.
- For each element $\frac{v_i}{N} + b$ find the nearest corresponding value in the domain D.
- substitute the values v_i with their quantised version q_i.

To get the de-quantised element $\hat{v}_i$ the operation is $\hat{v}_i = N(q_i - b)$. The obtained value is described as $\hat{v}_i$ since, usually, the return value has a certain amount of quantization noise.

Model Quantisation is a memory optimisation technique where model precision, from a typical 32-bit floating point precision, is reduced to a lower floating point representation. In deep learning systems, this precision reduction is extremely advantageous since it directly lowers the storage cost of parameters, activations, and gradients, while also reducing memory traffic, which is often a major bottleneck in modern accelerators. The appeal of FP16 comes from its simplicity in fact it preserves the floating-point structure of the model and therefore integrates easily into existing training and inference pipelines reaching good amount of memory savings. However, this simplicity, should not be confused with numerical equivalence to FP32. The main limitation of FP16 is its narrower dynamic range and lower precision, which make small gradient long accumulations vulnerable to rounding error. For this reason it's very often preferred a **Mixed Precision** strategy [36]. The standard execution consists of: storing master weights copy in 32-bit, kept mainly to avoid any type of gradient vanishing caused by low precision; this copy will be optimised during optim step; at each iteration, a 16-bit copy of the weights is used to perform a forward and backward pass, saving activations and gradients at 16-bit too. In [36] is also proposed a loss scaling to prevent small gradients from vanishing when represented in half precision. That design, actually, shows not only the importance of quantization for storage choice, but also as part of a broader numerical strategy in

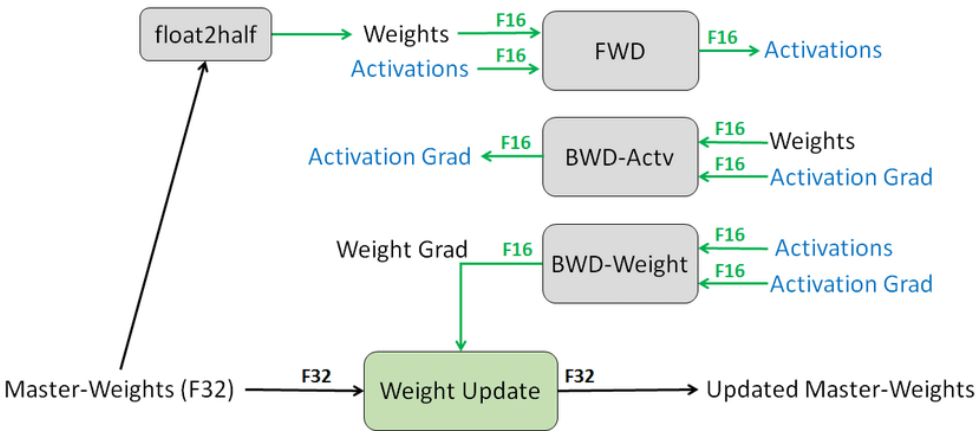


Figure 2.11: Mixed precision from [36]

which high precision is retained only where it is most necessary, while low precision is used where it yields the largest memory and throughput gains.

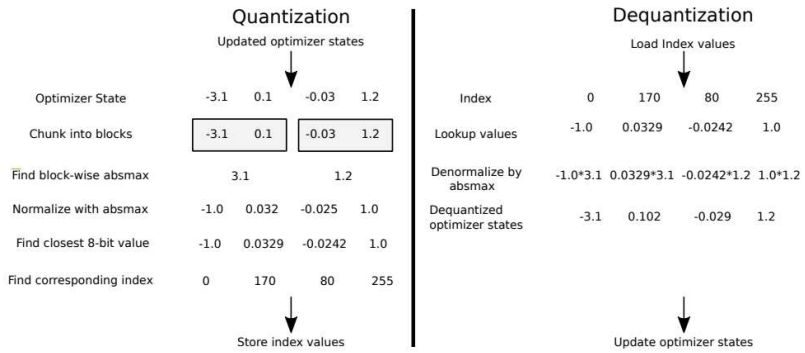


Figure 2.12: Block quantisation applied in [9]

The **Optimiser Quantisation** acts directly on the internal state of the optimiser. As said in 2.4, more complex optimizers require having a greater memory footprint, for example: a SGD with momentum requires one auxiliary state tensor per parameter, while Adam requires two, so when these states are stored in standard 32-bit precision they consume 4 bytes per parameter for momentum and 8 bytes per parameter for Adam, especially in large neural networks this becomes a major systems bottleneck: optimizer states can account for roughly 33-75% of total training memory, for example Adam states alone occupy 11 GB for the largest GPT-2 model and 41 GB for T5 [10]. This flaw makes 8-bit optimizers not a simple reformulation of the train pipeline, but a fundamental compressing strategy for improving learning capabilities. One of the main

breakthrough is represented by [10], showing that aggressive compression (8-bit) of optimizer states is possible without sacrificing the empirical behavior of standard 32-bit training. The process is composed of three main steps:

- **block-wise quantization**, which divides a state tensor into many smaller blocks and quantizes each block independently. The various advantages of this strategy are enhanced throughput, since blocks can be computed independently; outliers robustness since their effect influence only one block and, also, a better outliers approximation.
- **dynamic quantization** and, specifically, **Dynamic Tree Quantisation** [9] with a rework for non signed tensors.
- **stable embedding layer** to reduce the highly skewed token-frequency distribution that can induce large gradient variance in embedding parameters.

This method, at operational level, doesn't replace any optimisation rule, it just change how the optimiser states are stored, as also shown in the bitsandbytes documentation, the logic of the optimisation step is still performed in 32-bit while the tensor are quantised back to 8 bit after it. The empirical results reported by [10] are the main reason 8-bit optimizers became influential. Across a broad set of tasks, including GLUE fine-tuning with RoBERTa-Large, ImageNet classification, GPT-style language modeling, the 8-bit variants match or slightly surpass the replicated 32-bit baselines while reducing memory usage and slightly reducing total training time.

2.11.2 Gradient Accumulation

Gradient accumulation is a training strategy used when the desired global batch size does not fit into accelerator memory. Instead of performing an optimiser update after every physical batch, the model processes several smaller micro-batches sequentially, accumulates their gradients, and updates the parameters only once at the end of the accumulation window. In that sense, gradient accumulation is best understood as a mechanism for simulating large-batch optimisation under memory constraints rather than as a distinct optimisation algorithm.

3. Model and Dataset

In this chapter, two main components will be discussed in detail.

The first is the **Dataset**, including all the information related to its construction process, a thorough description of the various exploratory analyses, its internal structure, and, more generally, every aspect concerning the data gathering procedure. Particular attention will be devoted to explaining how the data were organized, prepared, their main characteristics emerged during the preliminary analysis phase.

The second component is the **Predictive Model**, with a detailed presentation of its architecture, the sub-models involved in the overall framework and the main choices made concerning the inference. This section will, therefore, provide a comprehensive overview of the modeling strategy, explaining deeply all the necessary information to fully understand the proposed architecture.

3.1 Dataset

Availability of an appropriate dataset, in **Supervised Learning**, is a one of principal requirement for the development and evaluation of efficient predictive models. This is even more decisive in regression problems, in particular in first buildign pahses, where the definition of the target variable and the selection of informative features is the main task in order to reach a representativeness of the collected observations. For these reasons a precise and detailed description is mandatory.

In the HPC context, and particularly in complex workload managers like SLURM, an extremely relevant problem is the optimisation of priority parameters, which are the main, and most important, factors in job waiting time; formally the delay between job submission and the execution start. For the specific purpose of predicting this metric, this dataset was constructed, mainly to demonstrate the ability of modern Transformer-based systems to internally capture and represent the complex dynamics underlying workload balancing in HPC environment. This demonstration could lead to various different developments in cluster optimisation:

- accurate prediction can be useful for users, who may rely on such estimations for planning workflow and management.

- for researcher interest in the behaviour of scheduler-controlled environments.
- to improve modern scheduling system, that uses simple filling technique (e.g. backfilling) to maximise the system utilisation, improving the precise evaluation of waiting time

However, despite the practical relevance of this task, the literature does not provide an established dataset specifically designed for SLURM job queue time prediction.

The purpose of this section is documenting the dataset construction process including the origin of the data, the definition of the target, the selected features and the main preprocessing decisions. All that is necessary both to justify the experimental setup adopted in the thesis, and to position the dataset as a reusable contribution for future work in the area.

3.1.1 Task Formulation

The task is formulated as a sequence-to-sequence regression problem. Let's consider

$$\mathcal{J} = (j_{-P}, \ldots, j_1, j_2, \ldots, j_T) \quad t_{-P}^{\text{sub}} < \cdots < t_1^{\text{sub}} < t_2^{\text{sub}} < \cdots < t_T^{\text{sub}} \tag{3.1}$$

the sequence of submitted jobs ordered by submission time, where j_i is the i-th job and t_i^{sub} is the relative submission time. Each job j_i is represented by a vector $\mathbf{x}_i \in \mathbb{R}^d$ containing all the available information: requested resources, requested walltime, partition, and other attributes related to the scheduling process. All the jobs j_i where $i \in [-P, 0]$ are defined as *context jobs*, are not prediction target and are used to give some cluster-status context.

The target associated with each job is its queue waiting time, defined as the elapsed time between submission and execution start:

$$\begin{aligned} y_i &= t_i^{\text{start}} - t_i^{\text{sub}} \\ \mathbf{y} &= (y_1, y_2, \ldots, y_T) \end{aligned} \tag{3.2}$$

The learning problem consists in estimating a function

$$f_\theta : (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T) \mapsto \hat{\mathbf{y}} \tag{3.3}$$

where $\hat{\mathbf{y}}$ denotes the predicted sequence.

With regard to the vector representing a job, in our case it is composed of three main subcomponents.

- The first is a textual component, which includes all the categorical, non-numerical job's attributes. Specifically, these features consist of the **Account Name**, **User Name**, **Quality of Service**, and the **Job Partition**. Although these variables are not numerical in nature, they carry important contextual information about the job and the user.

- The second is a resource-related component, which describes the set of resources requested by the job. This part includes the amount of **Memory**, expressed in megabytes, the **Nodes Number**, the number of **GPU** accelerators, the **CPUs Number**, and the **Wall-Time**. These factors together provide a numerical description of the computational resources needed for each job, which is essential for understanding its expected impact on the system's overall workload.

- The third component describe the temporal aspects, it is described by the main timestamps of the job lifecycle. This includes the **Submission Time**, the **Start Time**, and the **End Time**. In cases where the end time, or both the start time and end time, are not available at evaluation time, these values are masked. This masking strategy makes it possible to preserve a consistent input structure while explicitly accounting for missing temporal information during data processing and model handling.

3.1.2 Data Acquisition

The data was collected from the AImageLab Research Group's university cluster, a production system used primarily for research purposes. Given the "unique" nature of each system, whilst all the solutions applied in this specific case will be described in detail, this section should be viewed primarily as an execution pipeline that can be applied to various other clusters in order to develop specific models tailored to individual institutions.

To be more specific, the cluster is organised into three partitions

- **all_serial** exclusively used for debugging.

- **all_usr_prod** the main production partition, used to perform the majority of the tasks.

- **boost_usr_prof** the heavy train partition, used for training tasks that require more than 24 GB of VRAM.

Regarding users, they fall into the main categories of university actors: lecturers, researchers, PhD students and undergraduates.

For **Data Collection**, being this a non-high-sized cluster, it was possible to perform the gathering via **sacct**, a SLURM primitive, without any further solution. In this way it has been possible to select all relevant for accounting job metadata (and so allocation). To avoid any issues linked to inconsistent queues, we chose not to set the latest possible start date but instead utilised a 5-day grace period; for this reason, the data collected covers a period from 3 March 2025 to 19 February 2026, representing approximately 11 months of the cluster's operation. All of these informations have been stored in *JavaScript Object Notation* (JSON) format[24]. Once this comprehensive collection had been completed, it was decided to refine the collected data by removing:

- All data relating to cancelled and unstarted jobs, as these have a relatively minor impact on waiting times.
- Data relating to debug jobs in the all_serial partition, as, due to the cluster's structure, all those jobs are allocated on the login nodes and does not affect the other.
- The data from the last 5 and first 5 days of gathering for evaluation purposes, but only for context, again to avoid inconsistencies in the queues potentially caused by jobs not included in the collection.

From this refined set, more than the various terms relevant to the composition of the vector, described in subsection 3.1.1, were extracted. The ones extracted were: Job Id and Task Id used for table index; **account**; **user**; contraints; **qos**; **partition**; last state; priority; elapsed time, expressed in minutes; **submission time**; eligible time; **start time**; end time and all information about required resources. All the data was then organised into a table saved in *Parquet* format.

For **Sequential Representation**, however, it was sufficient to construct an indexed representation. For each validation job j_K, identified by its Job ID, the queue was generated by collecting all jobs that satisfied the inequality:

$$t_K^{\text{sub}} \leq t_i^{\text{end}} \wedge t_K^{\text{start}} \geq t_i^{\text{sub}} \quad \forall t_i \tag{3.4}$$

In that way, for the specific job j_K, all the contextual jobs are considered. In fact, during its waiting time, a specific job is influenced only by those jobs that are submitted before it starts, since after

that (except in exceptional cases not considered in this study), the job is already running, and jobs that end after its submission, trivially explained by the fact that any job already finished, cannot influeces a job not already submitted.

With regard to privacy, as this is an internal university thesis project, no substantive measures have been implemented. For different clusters, which may require more restrictive policies, it is recommended that the various usernames and accounts be modified in order to anonymise the dataset.

3.1.3 Target Variable

The target variable, formally defined as **Queue Time**, was actively derived from the dataset, since it was not, in itself, contained within the extracted data. The variable is defined as follows: for each job j_i belonging to the non-context jobs, the queue time is calculated as:

$$t_i^{\text{queue}} = t_i^{\text{start}} - t_i^{\text{sub}} \tag{3.5}$$

This variable is represented by a value $y \in \mathbb{Z}_0^+$ indicating the waiting time expressed in seconds. For evaluation and during training, in order to avoid excessive variance, this mesure is expressed in hours as will be explained in 3.1.4.

As mentioned in subsection 3.1.5, all jobs relating to cancellations and executions in the **all_serial** partition have been excluded. Apart from these, no other operations, such as clipping excessively high variables or removing fast-running jobs, were performed on the target variable. This decision is primarily due to the need for a comprehensive evaluator capable of handling even the most complex situations, such as instant-execution jobs or long SLURM vectors.

3.1.4 Feature Design

This section will cover the features used to represent the various jobs during training and validation. For a more schematic overview, refer to the table in 3.1.

As introduced in section 3.1.1, any job is represented by a vector $\mathbf{x}$, whose elements can be divided into three categories, which we shall call: **Textual Component**, **Resources Component** and **Temporal Component**.

Table 3.1: Job's features schema table

Feature	Type	Unit/Format	Available
account	String		yes
user	String		yes
qos	String		yes
partition	String		yes
submission time	Datetime	yyyy-MM-dd'T'HH:mm:ss	yes
start time	Datetime	yyyy-MM-dd'T'HH:mm:ss	depending
end time	Datetime	yyyy-MM-dd'T'HH:mm:ss	depending
walltime time	Integer	minutes	yes
memory	Integer	MB	yes
nodes	Integer	count	yes
CPU	Integer	count	yes
GPU	Integer	count	yes
queue	Integer	seconds	no

As for the **Textual Component**, this consists of all textual elements that represent categorical variables. It mainly contains the informations linked to the User context, all are represented by strings, and it's composed by:

- **User** and **Account** name, primarily needed to incorporate into the predictive model all the restrictions that the administrator has set at account and user level. They are also key feature to identify job vectors.
- **QoS** fundamental for scheduling since it has a significant impact on priority calculation and the preemption mechanism; furthermore, even in this case, there are specific resource constraints associated with it.
- **Partition** specifies the partition on which the job must run; this is a necessary feature in order, for the model, to understand how to represent the various subsets of the cluster.

The **Temporal Component** contains all the information relating to the job's temporal context, all the elements necessary for building sequences but also required to provide the temporal context for the model. All these elements are represented as Datetime objects, specified down to the seconds, and consists of **Submission**, **Start** and **End** time.

Lastly the **Resources Component**, which contains all user requests. This component includes both hardware requests and their timing, and consists of:

- **Walltime**, the maximum time asked for the job execution. It's represented in seconds by an Integer.
- **Memory**, the amount of memory required per node. It's represent the amount of MegaBytes xpressed by an Integer.
- **Nodes**, the required number of nodes. It's represented by an Integer.
- **CPU**, the required number of CPUs per node. It's represented by an Integer.
- **GPU**, the required number of GPUs per node. It's represented by an Integer.

The inclusion of this component is not merely a matter of priority, although this is necessary given that the required resources are factored into this calculation; it is also essential for understanding how to structure the cluster's internal resource system.

3.1.5 Preprocessing

The pre-processing stage wasn't the most time-consuming part; as we had collected the dataset ourselves, most of the features were already suitable for the purpose; nevertheless, it was necessary to carry out a few operations.

As regards the target variable, in order to reduce its variance, a change of units was applied, switching from a $\mathbb{Z}_0^+$ representation of seconds to a $\mathbb{R}_0^+$ representation of hours, this choice has been made mainly to reduce the strong variance of the dataset. Also the memory has been rescaled to GB.

In Temporal Component all elements are rescaled. Since it is not appropriate to represent them as dates, all time elements are represented as a Δ_t, relative to the earlier submission. To avoid too much elevate value this delta Δ_t is expressed in days.

To prevent information leakage, which could introduce waiting times directly into the model due to the simultaneous presence of Submission Time and Start Time, all Start Times and End Times that are earlier than Start Time among the evaluated job are masked with an impossible value, which is in this case -1.

The final pre-processing step, developed following some initial training attempts, was a response to the technical limitations of our systems. Although this will be explained in more detail in section 3.2, the system is primarily based on a main LLM block which, when handling long sequences, requires a significant amount of VRAM. In order to complete the training, we therefore limited it to only those examples whose queue did not exceed 250 jobs.

3.1.6 Dataset Statistics

Turning now to the more mathematical aspects of the dataset, the final version, following all the various steps described in the previous sections, consists of 104,000 samples; of these, 87,000 were used for the training process, whilst the remaining 17,000 were used for validation experiments.

Speaking about the target variable, the queue time, this takes the form of a distribution heavily concentrated around 0, which demonstrates that many jobs are able to start as soon as they are launched. Although some values are quite high and, as shown in the table 3.2, reach up to 60 hours, the mean, median and various percentiles are concentrated in ranges very close to 0.

Table 3.2: Job's features summary

	nodes	CPUs	GPUs	memory (GB)	walltime (days)	queue time (hours)
mean	1.0011	5.4801	1.0469	47.948	0.4971	0.7195
std	0.0546	5.8321	0.5947	40.996	0.6136	2.4077
25%	1.0000	1.0000	1.0000	20.000	0.0833	0.0000
50%	1.0000	4.0000	1.0000	32.000	0.3333	0.0013
75%	1.0000	8.0000	1.0000	64.000	1.0000	0.1900
min	1.0000	1.0000	0.0000	0.0078	0.0007	0.0000
max	4.0000	80.000	8.0000	768.00	3.0000	61.602

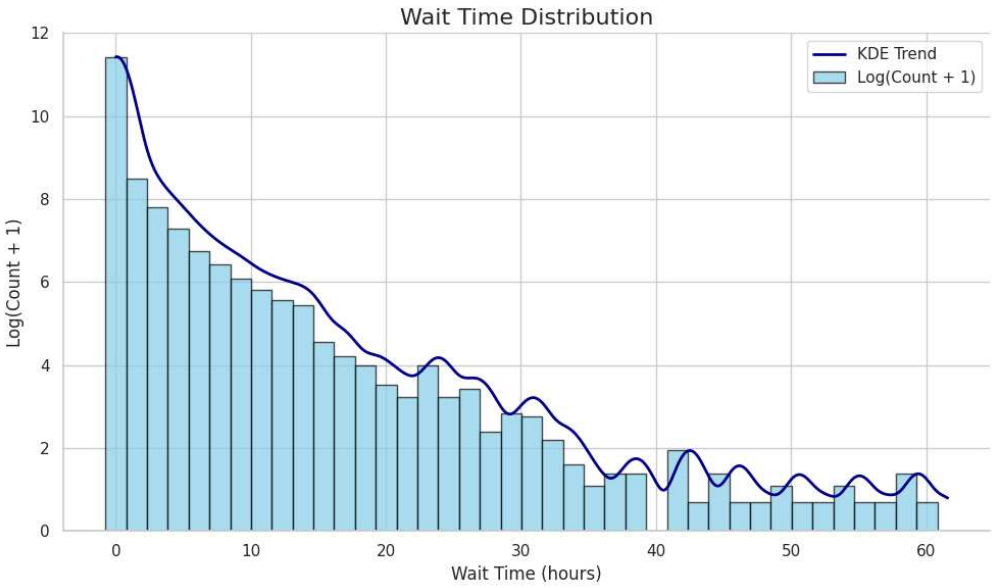


Figure 3.1: Log frequency of queue time

To reinforce the point made earlier, one may refer to Figure 3.1, which clearly shows the presence of a long tail in this distribution, particularly given that it is the logarithm of the frequency that is shown, not the frequency itself. Various analyses have also revealed a discrepancy in waiting times based on partition. The **boost_usr_prod** partition, in fact, is designed specifically for high-load workloads, which is precisely why fewer GPUs are made available in this case. It's possible that this uneven distribution of rsources is the cause of this discrepancy where waiting times are longer. This difference is shown in figure 3.2.

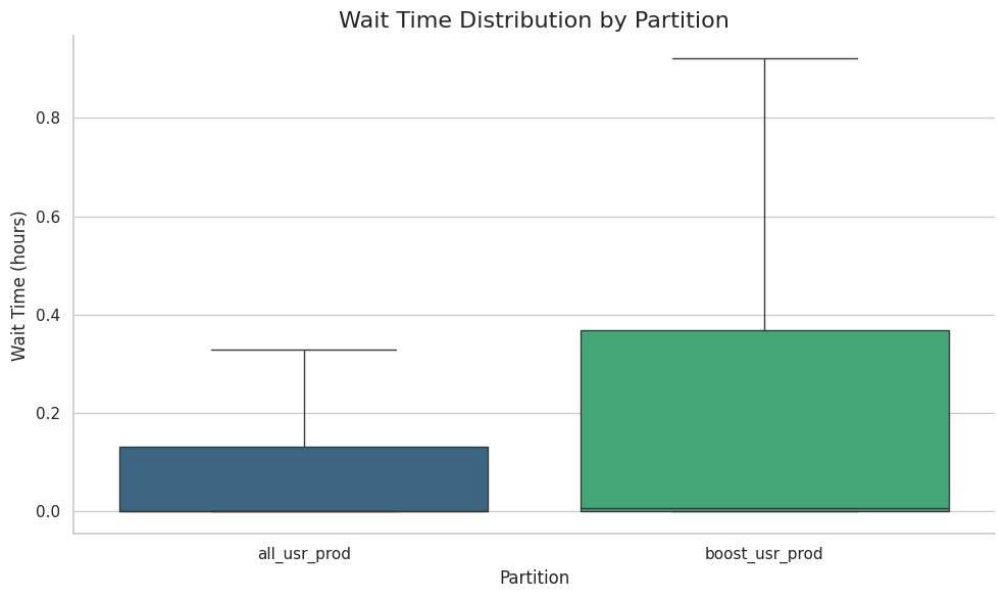


Figure 3.2: Boxplot base on partition

Looking at the various **numerical features**, it is clear that they're not extreamly correlated with the target one, as shown in the correlation matrix 3.3, whilst the relevant data can be seen in Table 3.2. Of all the correlations, the only ones worth noting, concern the hardware components, probably because, as one of the three components (GPU, CPU or memory) is increased, the others have inevitably to be upgraded as well, so as not to become a bottleneck in terms of computing performance. These are, also, linked to a small correlation with walltime, which can also be caused by practical considerations: as the number of resources increases, it is likely that we will be dealing with complex jobs that require long execution times.

As regards the distribution of these features, shown in figure 3.4:

- For **CPUs**, the graph shows not the exact number but its logarithm; this is because, typically, the number of CPUs required is a power of two. Apart from a large number of single-CPU jobs, and very few that exceed 32, the distribution is fairly even.
- **GPUs** are concentrated on a single unit, highlighting the fact that the AImageLab cluster primarily runs single-GPU jobs.
- An analysis of the **Walltime** shows a fairly scattered distribution across short and medium durations (<12 hours), with a final peak for jobs taking 24 hours. Jobs relating to upper

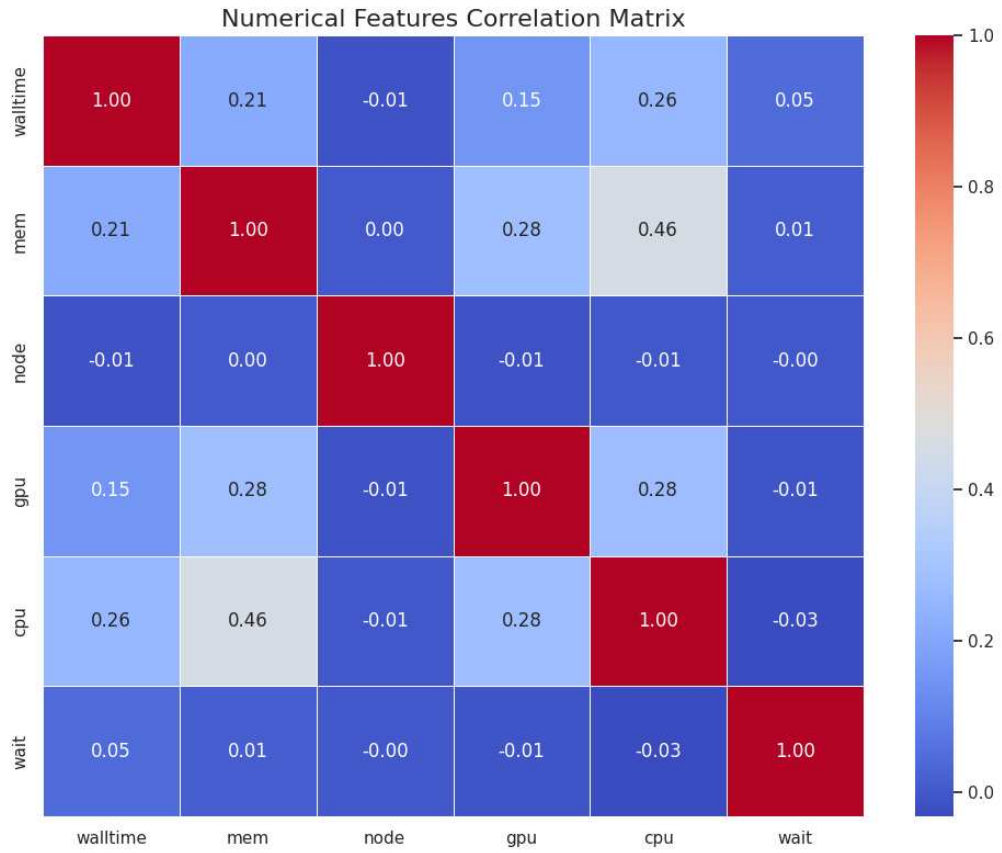


Figure 3.3: Correlation Matrix of numerical feature.

limits have not been included, as they are few in number and have virtually no impact. The scatter distribution is due, mainly, to the typical habit of selecting round or half-hourly times, a problem that not occurs when we deal with small time ranges (< 1 hour). After 12 hours, probably because of not so precise human prediction, the vast majority of walltime are set to 24 hours.

- Looking at the **Memory** usage, the distribution shows a peak at around 32 GB, which drops quite significantly down to around 100 GB; although there are jobs exceeding this threshold, their number is extremely small.

- **Nodes** distribution has not been plotted since almost all job, except marginal few ones, require only one node.

In conclusion, among the various relevant analyses, observations were made regarding the hourly and weekly workload. The graph of hourly submissions (Figure 3.5) shows a fairly typical traffic

(a) CPU

(b) GPU

(c) Walltime

(d) Memory

Figure 3.4: Numerical features distributions

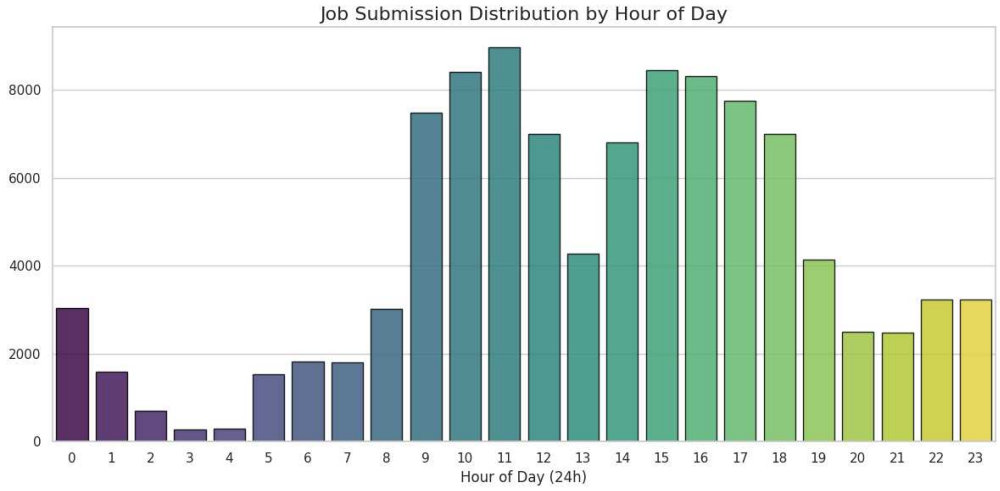


Figure 3.5: Hourly Job Submission

pattern, with high peaks during working hours, a drop during the lunch break, and low levels during the night. It is interesting to note, however, that between 11 pm and 1 am, traffic rises slightly before

levelling off again later in the night.

Looking at the weekly chart (Figure 3.6), we are also faced with a fairly typical situation: high workloads, which increase in intensity until the middle of the week, before slowly beginning to decline. Here too, there is a slight anomaly: on Sundays, despite being a day off, submissions to the cluster are actually higher than on Fridays, which is a working day.

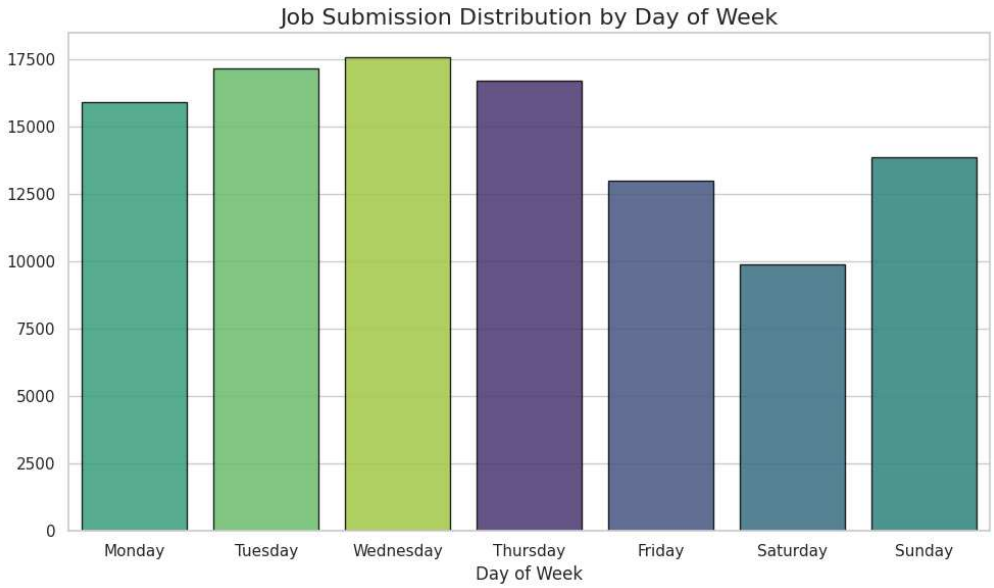


Figure 3.6: Weekly Job Submission

3.2 Proposed Model

The model consists of two main components: token projection and a predictive model. Token projection is the first component and is trained to generate different tokens from sub-components of the original vector. As for the predictive model, this is fine-tuned using LoRA adapters for the LLM (Llama[17], QWEN3[59], Ministral[32]) component; once the inference step is complete, a classification token is extracted and used to make the weather forecast.

3.2.1 Token Projection

Token projection, which algorithm is shown is Algorithm 1, is an intermediate solution that was developed after the difficulties of two other approaches became apparent:

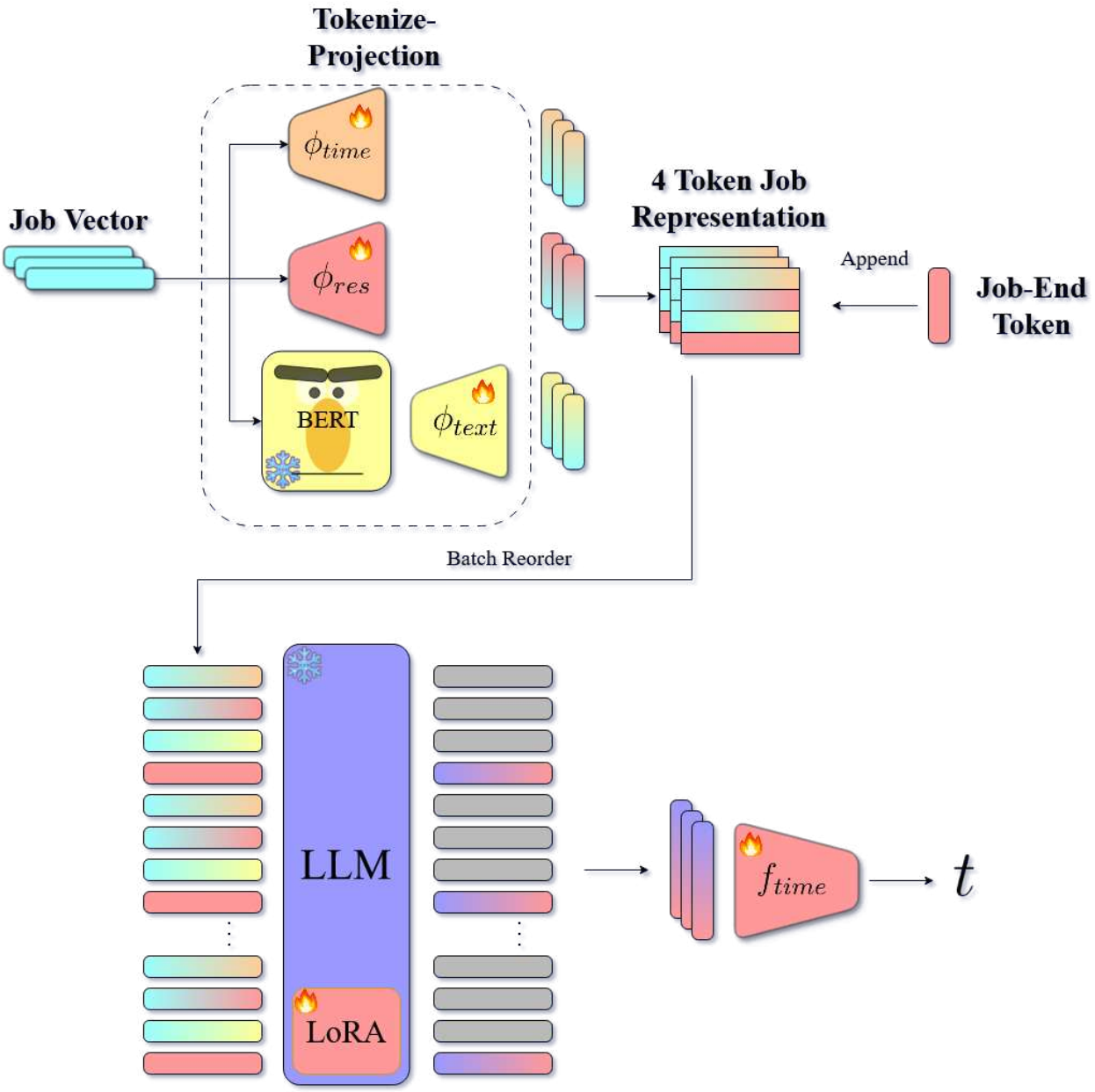


Figure 3.7: Queue-time prediction model

The first solution developed involved projecting all the components of the vector into a single token; although this was a more convenient and elegant approach, it had shown its learning limitations in the very first tests.

The second solution, on the other hand, simply "unrolled" the vector so that each feature could be projected into its own token and, using two special End Job and Start Job tokens, separated the various jobs in the sequence. This approach, however, had limitations in terms of processing

Algorithm 1 Token Projection

$x \rightarrow x_{text}, x_{res}, x_{time}$

$t_{res} \leftarrow \phi_{res}(x_{res})$

$t_{time} \leftarrow \phi_{time}(x_{time})$

$y_{\text{enc}} \leftarrow \text{BERT}(x_{text})$

$t_{text} \leftarrow \phi_{text}(y_{\text{enc}})$

$t_{job} \leftarrow \text{concat}([t_{res}, t_{time}, t_{text}, t_{end}])$ ▷ (Concat the tokens to get the job representation)

capacity; with 12 features per vector, plus the two special ones added, the number of tokens increased by an order of magnitude compared to the initial sequence.

In this context, the Token-Projection component comes into play: rather than fully unrolling the vector and projecting individual features, the vector is split into several logically related sub-vectors, which will be used for the tokens representing the jobs. These sub-components are the same as those presented in the previous section 3.1.4: Temporal, Textual and Resource components. A further modification, again applied to avoid an excessive increase in the number of tokens, was the removal of the special Start Job, retaining only the End Job.

Turning now to the components required for the projection:

- The Resource Projection ϕ_{res} consists of an MLP, with $n_{hidden} = 2$ with a hidden dimension $d = d_{LLM}/2$.
- The Temporal Projection ϕ_{time} consists of an MLP, with $n_{hidden} = 1$ with a hidden dimension $d = d_{LLM}$.
- The Textual Projection is composed by 2 element. The first is a BERT[12] encoder that, uses the 4 last hidden state to create the encoding, and another projection, which is a simple linear layer.

To provide BERT with a clearer context, the text features are not fed in their original form, but are reformulated into a string of the following type: “account:**<account>** user:**<user>** qos:**<qos>** partition:**<partition>**”.

This process must be carried out for all jobs in all batches across the various sequences, which makes training quite complex, as the batch sizes, which refer only to the number of sequences,

Table 3.3: All the main information about the model

	d_{in}	d_{out}	d_{hid}	n_{hid}
ϕ_{time}	3	d_{LLM}	d_{LLM}	1
ϕ_{res}	5	d_{LLM}	$d_{LLM}/2$	2
ϕ_{text}	d_{BERT}	d_{LLM}		
$ftime$	d_{LLM}	1	$d_{LLM}/2$	1
LoRA	r	α	p_{drop}	Proj
	16	32	0.05	Q, K, V, O

provide no information on the number of instances that this first part will need to process. Once the projection process is complete, the result will be three tokens representing information regarding: the resources required, the temporal context and the user context. At the end of this process the learned **Job-End Token** is concatenated to the group.

3.2.2 Time Prediction

The first step in the time series prediction uses a transformer-based system; in this case, various LLMs (Llama3[17], QWEN3[59], Ministral3[32]) were fine-tuned for being used as encoders, to ensure that the various job tokens take into account both their other components and the state of the cluster at the time of evaluation. This step is carried out by reordering all tokens so that they are in submission order; to perform this task, the system uses a mask that is provided as input to the model. To adapt the system for encoding purposes, fine-tuning was performed using LoRA[23] adapters, specifically following the QLoRA[11] process, which applies significant quantisation to the base model. The LoRA parameters used during training are as follows: $r = 16, \alpha = 32, p_{dropout} = 0.05$ and the adapters have been applied on all Q, K, V projections and the **output** one.

Once the tokens have run through the LLM, the Job-End tokens are extracted and used to predict queue time. The MLP f_{time} used for prediction has a single hidden layer and, in terms of internal dimensionality, has $d = d_{LLM}/2$; for the output, since the waiting times are always in the interval $[0, \infty]$, a function must be applied in order to map them to $\mathbb{R}^+$; for this reason is applied the function:

$$y = \log(1 + e^x)$$

allowing the function to be remapped continuously without having to use clamping systems or other non-continuous methods.

All the key information is summarised in Table 3.3

4. Results and validation

This chapter will analyse the final results of the various experiments and will be divided into two main sections: the first will focus on identifying the most suitable loss function for the task, whilst the second will concentrate on comparing different models to see how the system responds to changes in scale.

For the first section, the setup will be identical, with the sole difference of the loss function. For the model, as LLM component, will be used a Llama-3.2-1B; for the loss functions, instead, the two most common ones for regression tasks (MSE, MAE) will be used, along with a modified version of the Huber loss, which behaves in the opposite way: MSE for larger outputs and MAE for smaller ones.

The second section will focus on validate the model structure with various LLM component, finding the best setup for this regression task and analising the results for the chosen models. The selected LLM are: QWEN-3.5-0.8B and QWEN-3.5-2B [59]; Llama-3.2-1B [17] and Ministral-3-3B-Base-2512 [32].

4.1 Loss Analysis

Table 4.1: Various loss Results

	RMSE	MAE	R^2	$\text{least}_{10}\text{RMSE}$	$\text{top}_{10}\text{RMSE}$
Negative Huber	0.805	0.279	0.889	**0.002**	2.469
MAE Loss	1.056	0.276	0.810	0.174	3.312
MSE Loss	**0.720**	**0.212**	**0.911**	0.014	**2.240**

This section examines the impact that three different loss functions have on the predictive capabilities of the proposed model. This brief analysis is primarily due to the importance that various loss functions play in the training of regression models, particularly with regard to stability and generalisation ability; for this reason, it was deemed important to carry out this comparison.

To ensure a fair comparison, all experiments were conducted under the same conditions: the same architecture, batch size, learning rate, optimiser, and so on. Only the training loss was modified across the various scenarios. As regards the architecture, it was used a Llama-3.2-1B model, whilst all other components were described in the previous chapter. Regarding the analysis we consider **Mean Squared Error** (MSE), **Mean Absolute Error** (MAE), and a simple variant of **Huber Loss**. As stated in Section 2.3 Huber Loss tries to get the most of high error penalisation from MSE without too much instability, applying MAE to extremly high error. However, since the system in this case has numerous values close to zero, the MSE would lose effectiveness as it approaches that value, whereas with the MAE, smaller residuals are not subject to this decline. For this reason, a loss function with behaviour opposite to that of the Huber Loss was chosen. To easy identify this metrics, in reult table, is indicated as **Negative Huber** and it's equation is:

$$L_\delta(x) = \begin{cases} x^2 & \text{if} \quad |x| \geq 1 \\ |x| & \text{otherwise} \end{cases} \tag{4.1}$$

The trained models are evaluated using standard regression metrics: MAE, RMSE, and R^2, in order to capture complementary aspects of predictive performance. MAE gives an interpretable response on the average absolute prediction error, RMSE highlights the presence of large deviations by assigning greater weight to them, and R^2 measures the proportion of target variance explained by the model. Besides standard global metrics, we perform a more deep analysis by considering the mean RMSE over the top 10% highest-error samples and, conversely, over the 10% lowest-error samples, two percentile-based measures that provide additional information on the distribution of prediction errors on the easiest and hardest errors. Together, these measures help determine whether the model performance is uniformly distributed across the dataset or whether good average results are driven by strong predictions on only a subset of observations. These results are shown in Table 4.1.

The results show that MSE Loss leads to the best performance overall, except for the top_{10}RMSE. This suggests that strongly penalizing large prediction errors is beneficial for the queue-time prediction task, despite the low uniformity of dataset distribution. However, the excellent result for the least 10% of the Negative Huber demonstrates how the inclusion of an MAE term, when there are numerous values close to zero, can yield significant benefits during the inference process. A

clear imbalance in the results is evident across all three models, with the top 10% and the bottom 10% showing a marked difference in impact. These errors will be analysed in more detail below in section 4.3.

4.2 Model Comparison

Table 4.2: Model Comparison

	RMSE	MAE	R^2	$\text{top}_{10}\text{MSE}$	$\text{least}_1\text{0MSE}$	$\text{RAE}_{t>10}$
QWEN-3.5-0.8B-base	0.937	0.374	0.850	0.013	2.801	0.228
QWEN-3.5-2B-base	**0.678**	**0.196**	**0.921**	**0.006**	**2.118**	0.179
Llama-3.2-1B-base	0.720	0.212	0.911	0.014	2.240	**0.165**
Ministral-3-3B-base	0.732	0.231	0.908	0.016	2.273	0.208

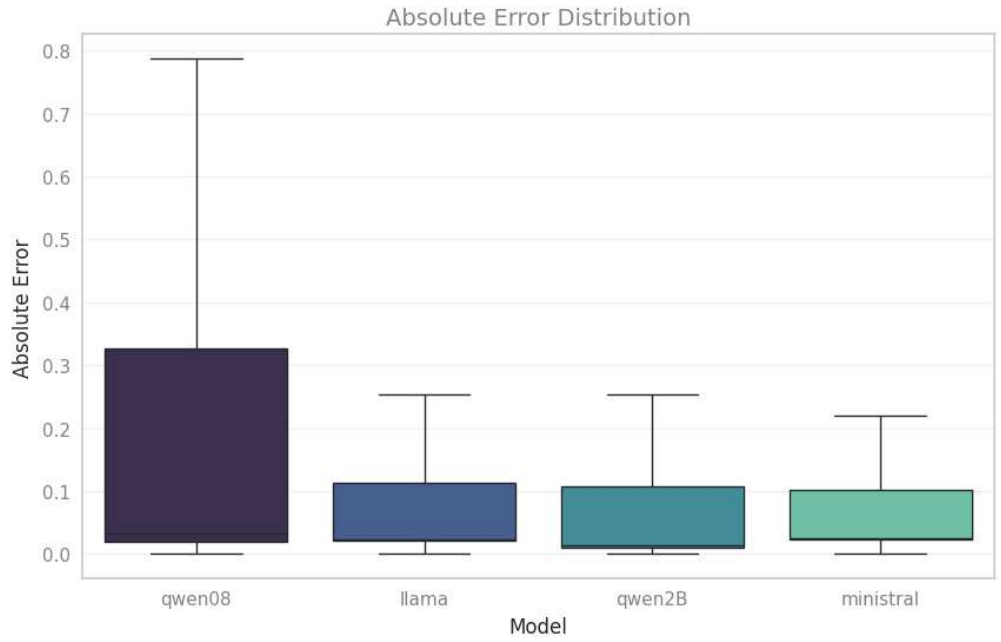


Figure 4.1: Absolute Error by Model

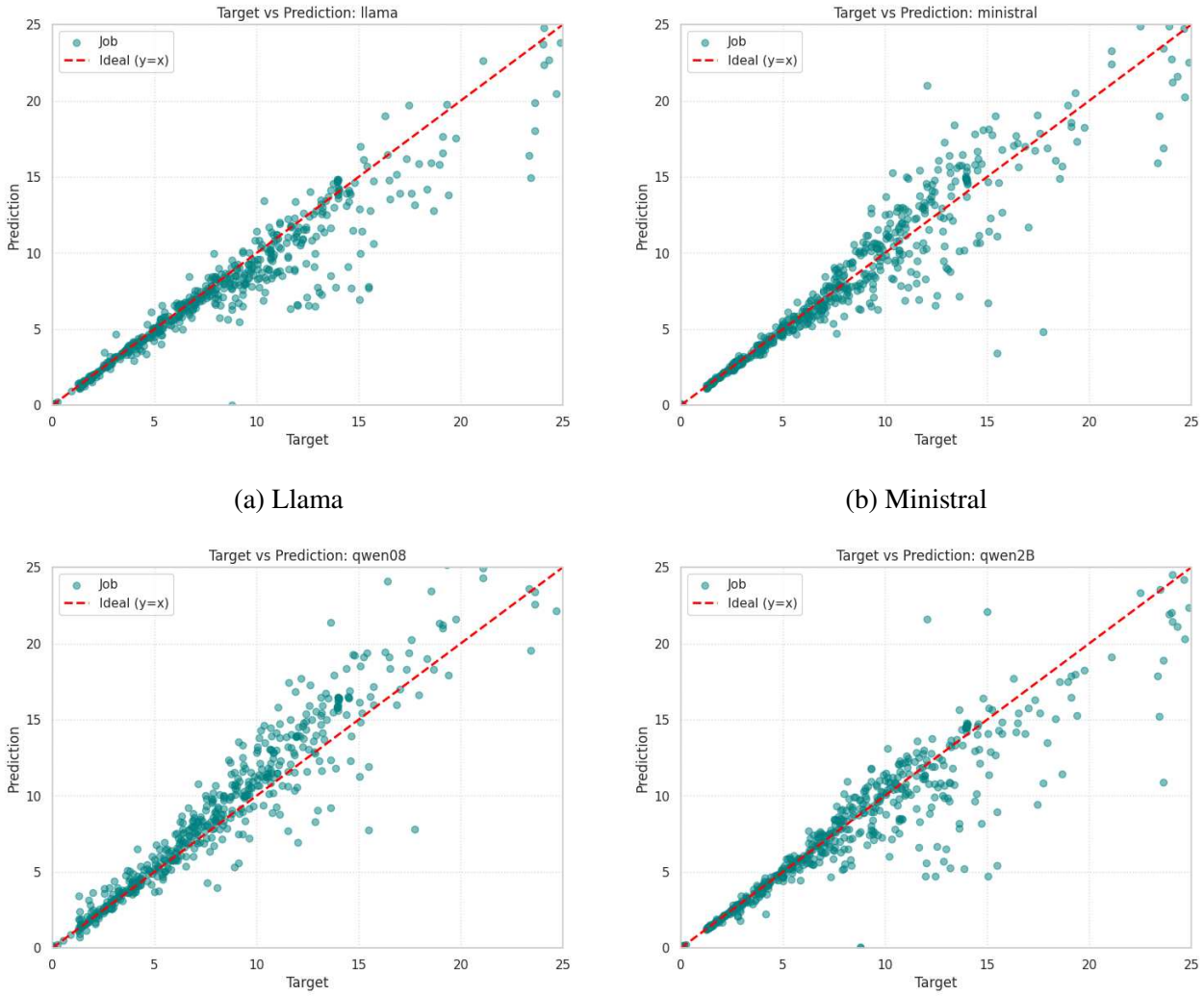
This section compares the considered models under a common evaluation framework. The aim is to highlight differences in performance across the metrics and to analise the various results

obtained by changing LLM scale. The models taken in account are: QWEN-3.5-0.8B and QWEN-3.5-2B [59]; Llama-3.2-1B [17] and Ministral-3-3B-Base-2512 [32]. The training, in this case too, followed the structure of the previous section. The setups analysed in Section 4.1 constituted an initial analysis to understand which choices would be most effective for this specific regression task. For this reason, once the optimal configuration had been determined by selecting the loss functions, the setup for comparing the models remained the same. Put simply, the entire previous section served to identify the appropriate setup for carrying out the experiments described here, which is, in this case, the MSE loss.

The main metrics have been reported in Table 4.2. The only difference is the presence of a Relative Absolute Error (RAE) for samples with target times > 10 hours, this is mainly due to unbalanced error along the dataset. As clear in the table one model managed to outperform all the other models in the comparison: QWEN-3.5-2B, which proved capable of achieving the highest scores across all the metrics, except one, assessed. From the same family comes the model with the poorest performance, QWEN-3.5-0.8B, which in fact performs worst across all metrics. This is entirely understandable, given that it is a model with fewer than a billion parameters; however, as this is the first time such a task has been proposed, it was necessary to observe the behaviour of a low-dimensional model. Although Qwen performed best, both Ministral and Llama proved to be solid contenders, achieving results that were very close to Qwen's and showing a clear gap between themselves and the bottom of the table. This striking difference is an initial indication that predicting queue times is no trivial matter; even a model with just under a billion parameters, optimised mainly for computational efficiency, begins to show significant deterioration compared to the other models.

4.3 Results

The first graph, shown in Figure 4.2 compares predicted queueing times with actual times. Here, one of the main challenges in training this system begins to emerge: the deterioration of predictions as waiting times increase. There may be various causes behind this issue, but the most obvious is certainly the inherent stochastic nature of the phenomenon. As mentioned in the section 3.1.6, a large proportion of **Walltimes** are not calculated with great precision; they are decided by users in a



(a) Llama

(b) Ministral

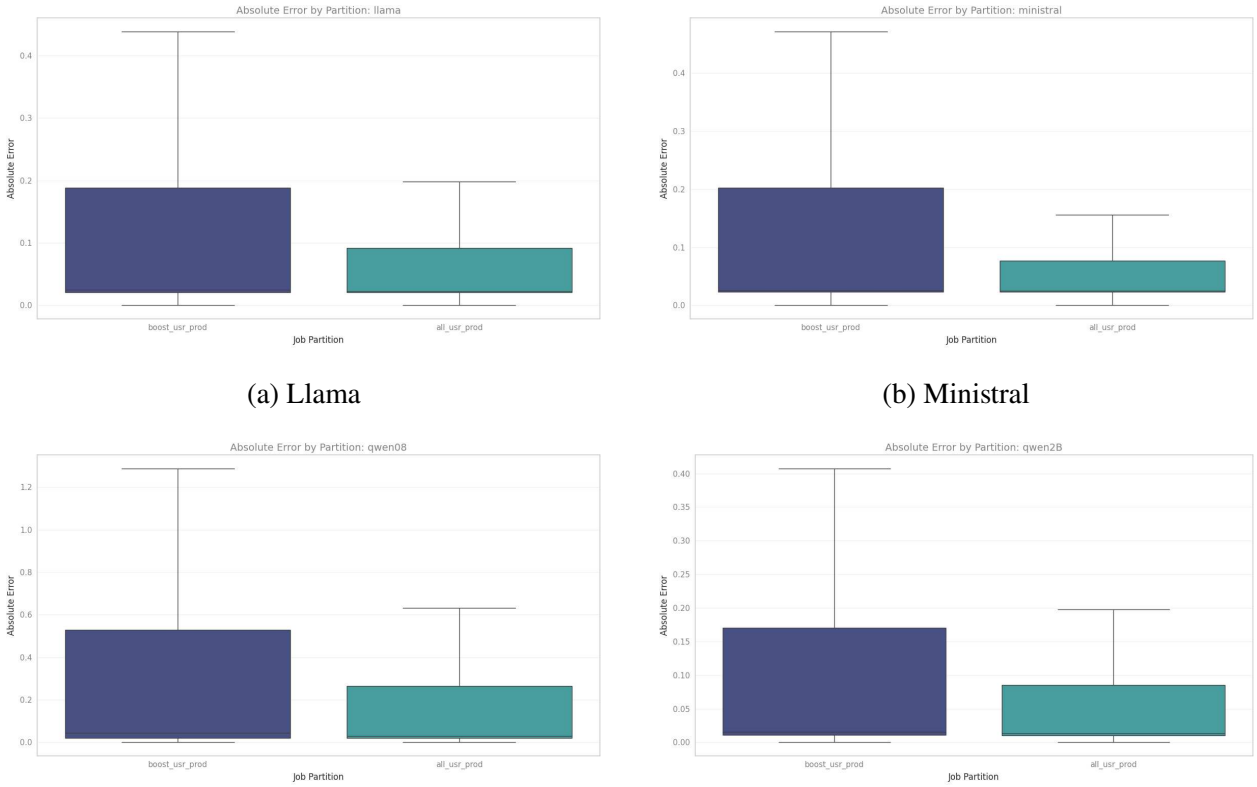
(c) Qwen 0.8B

(d) Qwen 2B

Figure 4.2: Target-Prediction plot

rather arbitrary manner (this is one of the reasons for their widely varying distribution) and, beyond a certain point, any assessment of the time is avoided and the wall time is set to the maximum limit. It's possible that exactly these increasingly inaccurate user estimation could undermine the scheduling system: let's consider a typical example of a high resource job j with a walltime t_j^{wall}, the schedule ensures that jobs entering backfilling have a wall time $t_k^{\text{wall}} < t_j^{\text{wall}}$; however, if this job is interrupted much earlier than predicted, this mechanism becomes partially ineffective, with higher-priority jobs finding themselves with a shorter waiting time than expected, as they will be able to occupy the resources allocated to them due to their higher priority, whilst smaller jobs will

be forced to wait longer as they cannot fully utilise the backfill mechanism.



(a) Llama

(b) Ministral

(c) Qwen 0.8B

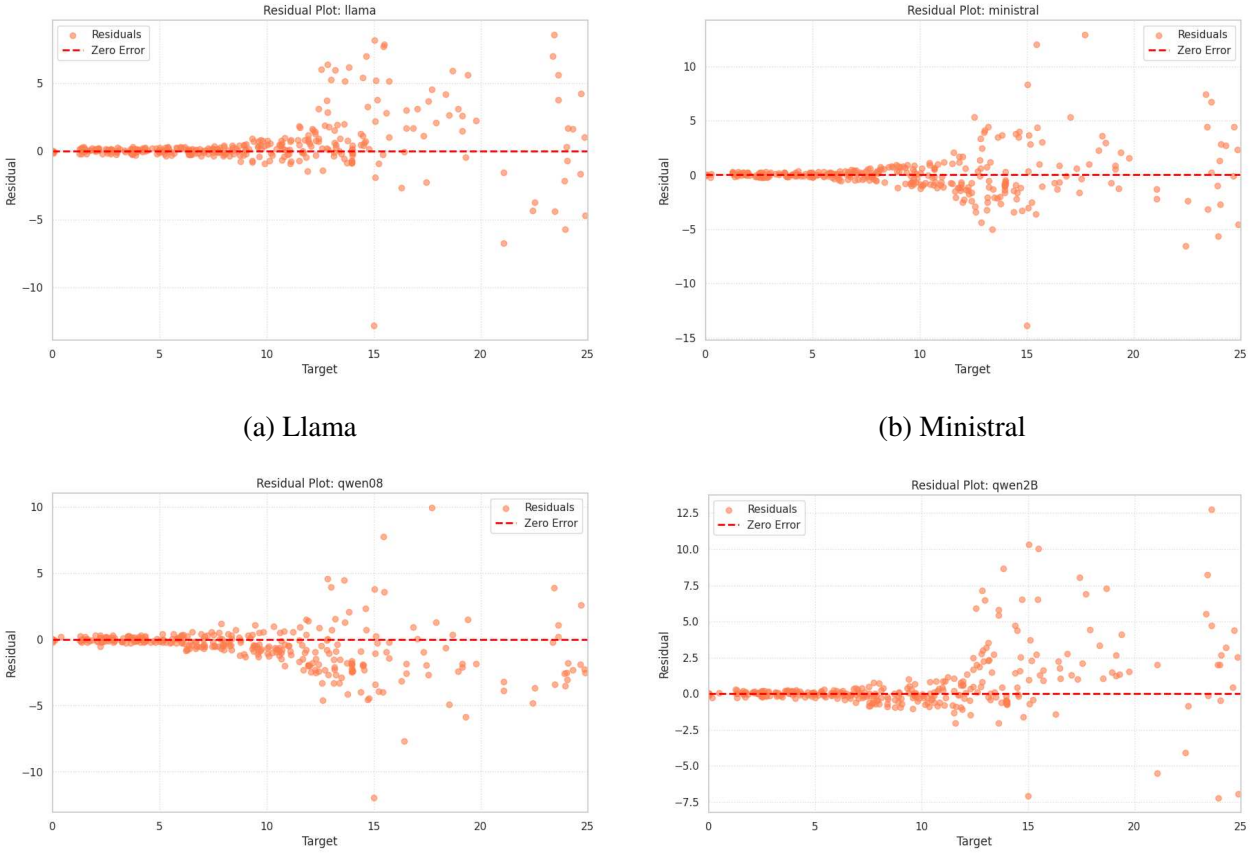
(d) Qwen 2B

Figure 4.3: MAE by partition

Secondly, but no less importantly, there is the issue of data scarcity; in fact, it is very difficult to find jobs with waiting times exceeding 5-10 hours. These are usually specific job vectors involving complex grid searches, very demanding tasks performed not so often.

To illustrate this point more clearly, one can observe the difference in prediction quality between the two partitions, Figure 4.5. It is evident that **boost_usr_prod**, which is specifically designed for more demanding tasks and therefore with a smaller presence within dataset, is the one that produces the poorest-quality predictions.

The residual plots, shown in Figure 4.5, also demostrate how, as the target variable increases, the error rises quite markedly. The funnel-shaped pattern clearly highlights the uneven distribution of data within the dataset. This phenomenon is known as *heteroscedasticity*, that is, a situation in which different populations within the system have different variances. This is most likely due to the timing issues described earlier.



(a) Llama

(b) Ministral

(c) Qwen 0.8B

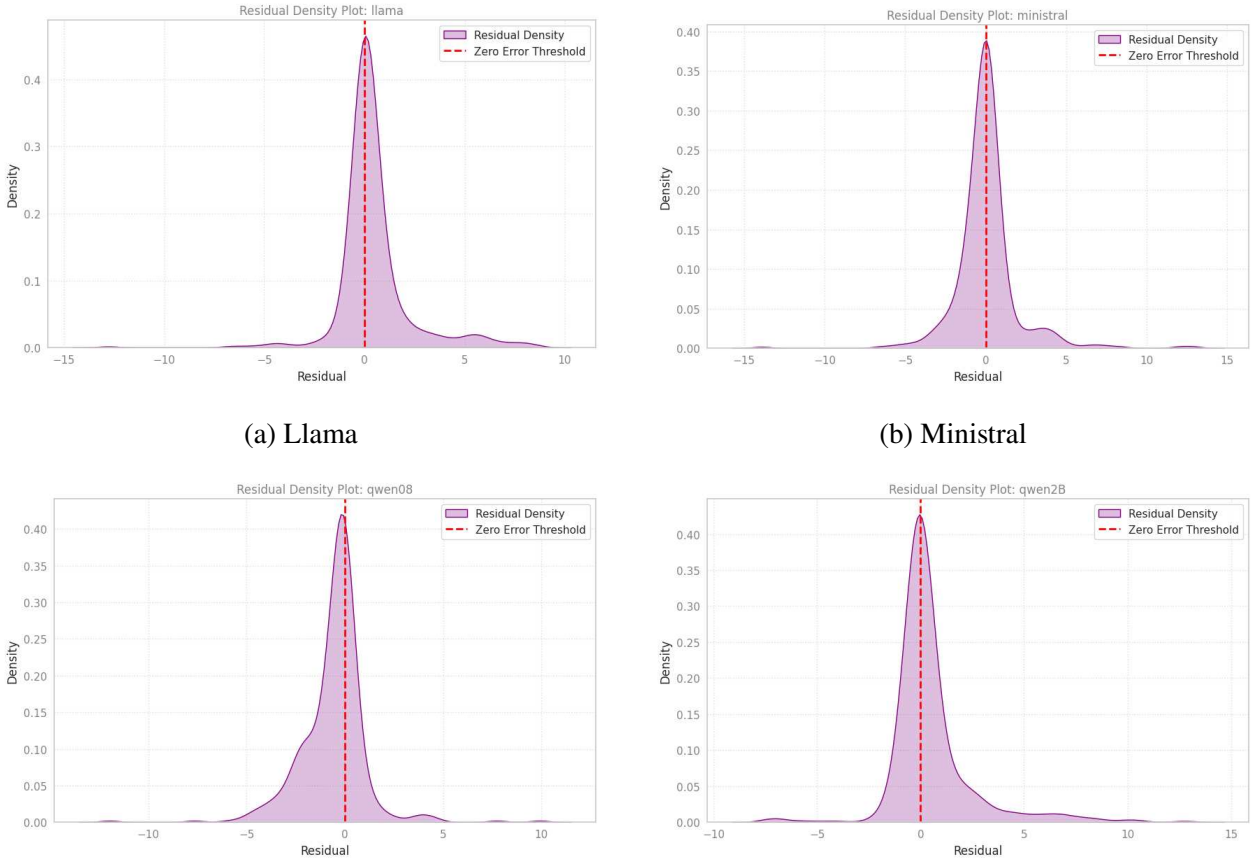
(d) Qwen 2B

Figure 4.4: Target-Resiudal plot

Regarding the distribution of residuals, the picture is fairly positive:

- centrality of the distribution suggests a model that is not subject to any particularly obvious biases.
- The distributions are fairly symmetrical, a sign that doesn't show any particular tendency to over or under estimate.
- Although the long tail exists, it does not have a particularly significant impact on the distribution. However, there are specific cases in which the prediction error is quite high (around 10 hours MAE)

As a final illustration of the model's difficulties in learning the dynamics of clusters for jobs with long queueing times, we can examine the Residual Range Distribution error, Figure 4.6, which



(a) Llama

(b) Ministral

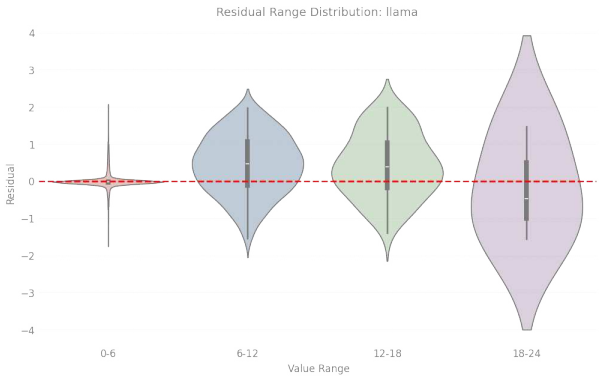
(c) Qwen 0.8B

(d) Qwen 2B

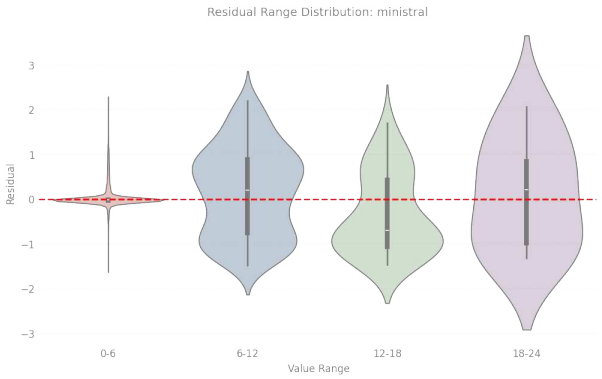
Figure 4.5: Resiudal distribution

highlights these difficulties more clearly when queueing times are relatively long.

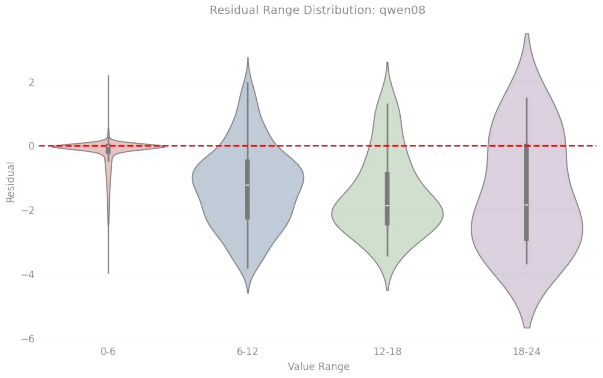
Despite the significant difference in performance, these results still fall within a reasonable range of acceptability. As shown in the results table, for predictions with a latency exceeding 10 hours, which we can consider a good threshold beyond which performance begins to decline, the relative error remains fairly limited and reaching, in the best case with Llama, 16%.



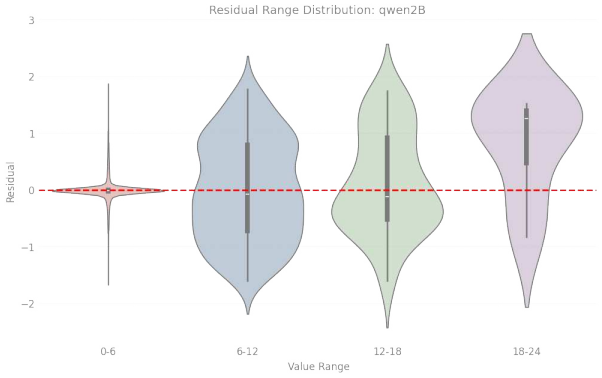
(a) Llama



(b) Ministral



(c) Qwen 0.8B



(d) Qwen 2B

Figure 4.6: Resiudal Range Distribution

5. Conclusions and Future Works

5.1 Conclusion

In this work, we have focused on exploring initial approaches to optimising large HPC clusters, using solutions centred on intelligent systems, rather than the usual empirical methods. Initially, we focused on building a dataset suitable for our objectives, collecting and analysing data from our university cluster at AImageLab, a production cluster specifically designed for research purposes. To better explain the composition of this cluster's overall workload, we also described the preliminary analyses carried out on the system and, most importantly, the method and rationale behind the choice of features: a crucial step in describing not only a dataset, but also a framework upon which to base the consistent reapplication of this evaluator to other systems.

We then turned our attention to building a model suitable for the task of predicting queue waiting times. The main feature of the architecture is the two-stage inference process: in the first stage, which focuses on individual jobs without an overall view, three semantically related tokens are extracted from each job; this approach was chosen as an efficient middle ground between **unrolling the vector** into various *feature-tokens* and a simple transformation from job to token; in the second phase, however, by exploiting the ability of transformer-based systems to evaluate long chains of dependencies, we perform the main evaluation of waiting times.

Finally, we carried out a series of experiments, divided into two main phases. In the first one, we conducted research to determine the optimal configuration for this inference system, specifically with regard to the loss function which, in regression systems, is one of the most significant factors as it alters the statistical metric evaluated during training. For the second phase, instead, we focused on comparing various transformer architectures of different dimensions to observe their behaviour in this specific task. The results obtained were nevertheless promising, demonstrating the system's excellent ability to assess waiting times in the low to medium ranges (i.e. from instantaneous tasks up to jobs with a 12-hour wait); on the other hand, however, performance shows a significant drop for jobs with the longest waiting times, a predictable situation, especially considering both the small number of jobs in this situation and all the internal dynamics within the cluster that lead to a

significant increase in variance for these long-waiting jobs (i.e. poor timing estimates by the user, crashes, and so on). Despite the significant disparity in terms of errors, however, those in the top tier remain relatively limited, as demonstrated by the RAE results.

5.2 Future Works

As mentioned in the introduction to this thesis, this field of research remains a largely unexplored niche, so there are countless opportunities for development. A first and necessary step is certainly the development of strategies to reduce errors on long-waiting jobs; as noted, these are the primary cause of the deterioration in metrics within the system: more comprehensive studies of phenomena within clusters, particularly regarding long dependencies, could help develop an ad hoc solution to the problem. Another way to mitigate this phenomenon, rather than focusing on studying it, could employ a more "brute-force" approach by applying robust data augmentation strategies, thus allowing the data alone to incorporate the management of the phenomenon into the model. Unfortunately, in such an interdependent system, the search for good enough strategies is by no means straightforward and, indeed, may require considerable research effort to develop solutions capable of fully capturing these phenomena and compensating for the lack of in-depth knowledge.

A further, even more interesting development would involve the possibility of incorporating scheduling weights directly into the models. In our case, we have focused on prediction for a system that is now stable; a model that is certainly useful, but which loses its significance if the cluster undergoes too many changes over time. Having systems, perhaps through cross-attention or more complex systems, that can adapt according to the current Slurm configuration would, first and foremost, be much more useful in the long term, but would also allow for the development of even better solutions regarding cluster optimisation: faster simulations for evaluating the optimal configuration or, more complex but undeniably useful, "actor-critic" [16] strategies to completely replace priority calculation with learning models capable of adapting to the system by minimising a specific metric. Despite their usefulness, however, as these are highly complex systems that also involve significant financial risk, finding methods to train them would be even more challenging given the lack, both in the literature and in practical terms, of datasets containing these numerous configuration variations. It is precisely in this field that existing simulating solutions in the literature

can be utilised: strategies to integrate training processes with modern simulators could pave the way for solutions to analyse configuration variations without causing any damage to production systems.

These are just a few of the possible developments; cluster optimisation is a complex task which, despite the challenges, will become increasingly prevalent in the coming years as the demand for computational resources grows, thereby opening the door to ever more efficient solutions.

Bibliography

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. 2020. arXiv: 2012.13255 [cs.LG]. URL: https://arxiv.org/abs/2012.13255.

[2] Joshua Ainslie et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245 [cs.CL]. URL: https://arxiv.org/abs/2305.13245.

[3] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". In: 30 (1967), pp. 483–485.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

[5] Jinze Bai et al. *Qwen Technical Report*. 2023. arXiv: 2309.16609 [cs.CL]. URL: https://arxiv.org/abs/2309.16609.

[6] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. English. Publisher Copyright: © 2023 Elsevier Inc. All rights reserved. Netherlands: Elsevier, Jan. 2022. ISBN: 9780128141212. DOI: 10.1016/C2016-0-04273-0.

[7] Rajkumar Buyya, Toni Cortes, and Hai Jin. "An Introduction to the InfiniBand Architecture". In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. 2002, pp. 616–632. DOI: 10.1109/9780470544839.ch42.

[8] Augustin Cauchy. "Méthode générale pour la résolution des systèmes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25 (1847), pp. 536–538.

[9] Tim Dettmers. *8-Bit Approximations for Parallelism in Deep Learning*. 2016. arXiv: 1511.04561 [cs.NE]. URL: https://arxiv.org/abs/1511.04561.

[10] Tim Dettmers et al. *8-bit Optimizers via Block-wise Quantization*. 2022. arXiv: 2110.02861 [cs.LG]. URL: https://arxiv.org/abs/2110.02861.

[11] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG]. URL: https://arxiv.org/abs/2305.14314.

[12] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: (2019). arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[13] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[14] Jeffrey L. Elman. “Finding structure in time”. In: *Cognitive Science* 14.2 (1990), pp. 179–211. ISSN: 0364-0213. DOI: https://doi.org/10.1016/0364-0213(90)90002-E. URL: https://www.sciencedirect.com/science/article/pii/036402139090002E.

[15] Jerome H. Friedman. “Greedy Function Approximation: A Gradient Boosting Machine”. In: *Annals of Statistics* 29 (2000), pp. 1189–1232.

[16] Majid Ghasemi, Amir Hossein Moosavi, and Dariush Ebrahimi. *A Comprehensive Survey of Reinforcement Learning: From Algorithms to Practical Challenges*. 2025. arXiv: 2411.18892 [cs.AI]. URL: https://arxiv.org/abs/2411.18892.

[17] Aaron Grattafiori et al. “The Llama 3 Herd of Models”. In: (2024). arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[18] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Commun. ACM* 31.5 (May 1988), 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: https://doi.org/10.1145/42411.42415.

[19] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[20] Geoffrey Hinton. *rmsprop: Divide the gradient by a running average of its recent magnitude*. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[21] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359 –366. ISSN: 0893-6080. DOI: http://dx.doi.org/10.1016/0893-6080(89)90020-8. URL: http://www.sciencedirect.com/science/article/pii/0893608089900208.

[23] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[24] *ISO/IEC 21778:2017 Information technology — The JSON data interchange syntax*. ISO. 2017.

[25] Robert A. Jacobs et al. “Adaptive Mixtures of Local Experts”. In: *Neural Computation* 3.1 (Mar. 1991), pp. 79–87. ISSN: 0899-7667. DOI: 10.1162/neco.1991.3.1.79. eprint: https://direct.mit.edu/neco/article-pdf/3/1/79/812104/neco.1991.3.1.79.pdf. URL: https://doi.org/10.1162/neco.1991.3.1.79.

[26] M Jette and M Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: (Dec. 2002). URL: https://www.osti.gov/biblio/15002533.

[27] Mandar Joshi et al. *SpanBERT: Improving Pre-training by Representing and Predicting Spans*. 2020. arXiv: 1907.10529 [cs.CL]. URL: https://arxiv.org/abs/1907.10529.

[28] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[29] Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005. ISBN: 0521608279. URL: http://www.amazon.de/Quantile-Regression-Econometric-Society-Monographs/dp/0521608279/ref=sr_1_1?ie=UTF8&qid=1312553603&sr=8-1.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems* 25 (2012). Ed. by P. Bartlett et al., pp. 1106–1114. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[31] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: 1909.11942 [cs.CL]. URL: https://arxiv.org/abs/1909.11942.

[32] Alexander H. Liu et al. *Ministral 3*. 2026. arXiv: 2601.08584 [cs.CL]. URL: https://arxiv.org/abs/2601.08584.

[33] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: 1907.11692 [cs.CL]. URL: https://arxiv.org/abs/1907.11692.

[34] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101.

[35] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[36] Paulius Micikevicius et al. *Mixed Precision Training*. 2018. arXiv: 1710.03740 [cs.AI]. URL: https://arxiv.org/abs/1710.03740.

[37] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE Standard for Binary Floating-point Arithmetic*. ANSI/IEEE / American national standards institute/Institute of electrical and electronics engineers. IEEE, 1985. URL: https://books.google.it/books?id=3L0gAQAAIAAJ.

[38] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781.

[39] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155.

[40] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL]. URL: https://arxiv.org/abs/1802.05365.

[41] Ofir Press, Noah A. Smith, and Mike Lewis. *Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation*. 2022. arXiv: 2108.12409 [cs.CL]. URL: https://arxiv.org/abs/2108.12409.

[42] Alec Radford et al. *Improving Language Understanding by Generative Pre-Training*. 2018. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

[43] Rafael Rafailov et al. *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. 2024. arXiv: 2305.18290 [cs.LG]. URL: https://arxiv.org/abs/2305.18290.

[44] Albert Reuther et al. "Scheduler technologies in support of high performance data analysis". In: (Sept. 2016), 1–6. DOI: 10.1109/hpec.2016.7761604. URL: http://dx.doi.org/10.1109/HPEC.2016.7761604.

[45] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[46] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[47] Ryan Cox and Levi Morrison. *Fair Tree Fairshare Algorithm for Slurm*. https://slurm.schedmd.com/SC14/BYU_Fair_Tree.pdf. Accessed: 2026-03-11. 2014.

[48] SchedMD LLC. *Slurm Workload Manager: Classic Fair-Share Algorithm*. https://slurm.schedmd.com/classic_fair_share.html. Accessed: 2026-03-11. 2026.

[49] SchedMD LLC. *Slurm Workload Manager: Fair Tree Fair-Share Algorithm*. https://slurm.schedmd.com/fair_tree.html. Accessed: 2026-03-11. 2026.

[50] Zhihong Shao et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. 2024. arXiv: 2402.03300 [cs.CL]. URL: https://arxiv.org/abs/2402.03300.

[51] Noam Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017. arXiv: 1701.06538 [cs.LG]. URL: https://arxiv.org/abs/1701.06538.

[52] Nikolay A. Simakov et al. "A Slurm Simulator: Implementation and Parametric Analysis". In: (2018). Ed. by Stephen Jarvis, Steven Wright, and Simon Hammond, pp. 197–217.

[53] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864.

[54] Yunpeng Tai. *A Survey Of Regression Algorithms And Connections With Deep Learning*. 2021. arXiv: 2104.12647 [cs.LG]. URL: https://arxiv.org/abs/2104.12647.

[55] R. Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society (Series B)* 58 (1996), pp. 267–288.

[56] Ashish Vaswani et al. "Attention Is All You Need". In: (2023). arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[57] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[58] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652.

[59] An Yang et al. *Qwen3 Technical Report*. 2025. arXiv: 2505.09388 [cs.CL]. URL: https://arxiv.org/abs/2505.09388.