Defence, Safety & Security
www.tno.nl
+31 88 866 00 00
info@tno.nl

GPTNL-DEL-4002-[1.0] – 22 December 2025

System Architecture Document

Training Pipeline

Author(s)	Julio A. de Oliveira Filho [Editor] Claartje Barkhof [Editor] Andrei Roncea [Editor] Athanasios Trantas Martino Mensio Alexandru Turcu Thomas van den Osch Simone van Bruggen Erik de Graaf
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	109
Number of appendices	0
Project name	GPT-NL
Project number	060.58424

Contents

1 Introduction

The GPT-NL project aims to develop a Dutch-English large language model (LLM) from the ground up to promote technological sovereignty and strengthen the Dutch and broader European LLM ecosystem. Achieving this objective requires a structured systems engineering approach encompassing requirement's elicitation, design, implementation, and validation. Beyond the creation of the model itself, sovereignty and community growth depend on transparent dissemination of knowledge about how such a system is built. This document therefore presents the architectural blueprints—both in code and documentation—for the second part of this development phase: the **System Architecture of the Training Pipeline**.

The documentation and systematic management of this technological blueprint are intended to stimulate new research directions and enable future improvements. The **GPT-NL System Architecture** effort serves as the foundation for these goals by providing a coherent, well-documented engineering framework for large-scale model development.

From a general point of view, the system architecture activities provide a structured conceptual model defining the organization, behavior, and interactions of system components. It offers a high-level view of how hardware, software, data, and processes collaborate to achieve the intended system goals. Through clear specification of components, interfaces, and design principles, the architecture ensures that key system attributes—such as performance, scalability, security, and maintainability—are addressed systematically and in alignment with stakeholder requirements and operational constraints.

Within the GPT-NL team, system architecture plays a coordinating role by providing a shared technical framework that guides design, implementation, and verification across teams. This work, conducted under **Work Package 13 (WP13)**, facilitates communication among engineers, researchers, and developers by defining clear interfaces and dependencies. The architectural team ensures design consistency, manages technical risks, and balances trade-offs among quality attributes. As a result, this document and the associated work contribute to the alignment of strategic objectives and technical execution, promoting system coherence, continuity, and effective integration throughout the development lifecycle.

The overview of the processes, tasks, and artifacts related to the architectural work is depicted in Figure 1. The system architecture team collaborates with all other working packages, but closest with **WP12 (Data Curation)**, **WP14 (Model Development)**, and **WP18 (Data Acquisition and Quality)**. While WP12 and WP14 lead algorithmic development—such as the selection of filters, models, and training techniques—WP13 focuses on translating these designs into structured, maintainable, and scalable code. This includes defining clean interfaces between modules, ensuring continuous data processing flows suitable for HPC environments, and addressing non-functional aspects such as security, documentation, and energy efficiency. The WP18 is responsible for the processes of contacting data providers and acquiring/creating datasets. They are strongly involved with the architecture team assessing the quality of the data during and after the curation phase.

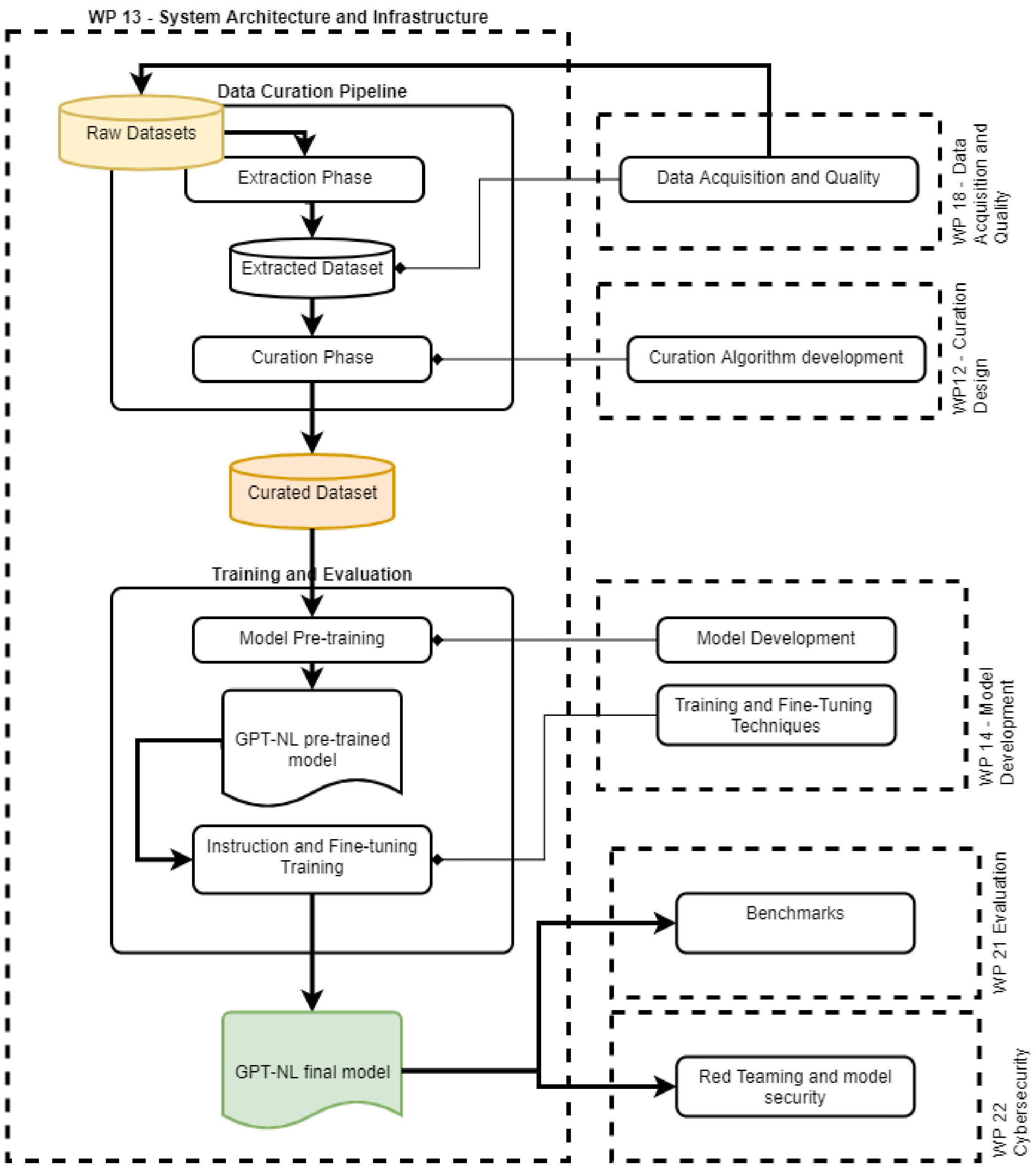

WP 13 - System Architecture and Infrastructure
Data Curation Pipeline
Raw Datasets
Extraction Phase
Extracted Dataset
Curation Phase
Curated Dataset
Training and Evaluation
Model Pre-training
GPT-NL pre-trained model
Instruction and Fine-tuning Training
GPT-NL final model
Data Acquisition and Quality
WP 18 - Data Acquisition and Quality
Curation Algorithm development
WP12 - Curation Design
Model Development
Training and Fine-Tuning Techniques
WP 14 - Model Development
Benchmarks
WP 21 Evaluation
Red Teaming and model security
WP 22 Cybersecurity

Figure 1: Overview main GPT-NL Processes and Work Packages

In general, LLM development can be divided into two main components: **data curation** and **model training and validation**. These components differ significantly in their technical focus and data processing requirements.

- The **data curation pipeline** encompasses all processes from data acquisition to the creation of a uniform dataset ready for model training. This includes systematic reasoning and documentation of inclusion and exclusion criteria, as well as the production of standardized datasets for both training and public release. The data curation pipeline is subdivided in two phases: the **data extraction phase** and the **data curation phase**. The whole curation pipeline and its phases are detailed in the next sections. Architectural artifacts from this pipeline include:

  - Software developed for data acquisition, extraction, curation, and dataset deployment.
  - Documentation of third-party software and hardware stacks—such as DataTrove, PrivateAI, and SURF's Snellius—including configuration details, versioning, and integration procedures.
  - CI/CD frameworks for testing, logging, and evaluating both the platform and resulting datasets.
  - Records of architectural decisions, design rationales, and supporting technical documentation.
  - Security, privacy, and energy monitoring mechanisms for development and operational phases.
  - Final technical reports and communication materials, including this document and supporting white papers.
- The **model training and validation phase** includes data preparation, tokenization, model pre-training, instruction fine-tuning, and performance evaluation. It results in a standardized and reproducible model package for internal use and community release. Artifacts from this phase include:

  - Software for data mixing, tokenization, model training, fine-tuning, and deployment.
  - Documentation of third-party stacks such as OlMO and the Snellius HPC infrastructure, detailing configurations and integration.
  - CI/CD support for testing and performance tracking.
  - Documentation of design decisions, system rationale, and supporting non-functional design considerations.
  - Security, privacy, and energy monitoring tools.
  - Final deliverables, including technical documentation and dissemination materials.

This document, **System Architecture Document – Training Pipeline**, covers the GPT-NL model training. Details on data curation are presented in the related document: **System Architecture Document – Data Curation Pipeline**[1]. As introduction, we present in the following the architectural overview of the Training Pipeline.

1.1 Architectural Overview of the GPT-NL Training Pipeline

The GPT-NL training pipeline transforms curated data into deployment-ready language models optimized for Dutch and English language tasks. The focus tasks for GPT-NL are summarization, simplification, and question answering based on contextual information (commonly known as Retrieval Augmented Generation pipelines). It consists of two major phases: **pre-training** and **instruction fine-tuning**. Each of these phases has distinct architectural requirements, computational demands, and outputs.

[1] TNO, GPT-NL Project, Report; *GPTNL-DEL-4001-1.0-System Architecture Document – Data Curation Pipeline*, December 2025.

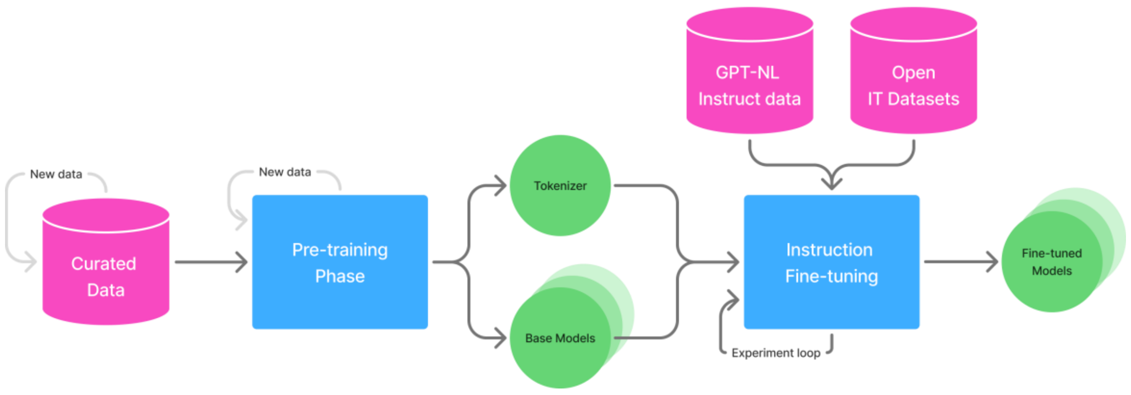


Figure 2: Introduction GPT-NL Training Process

The training pipeline starts after the data curation pipeline[1]. It consumes curated datasets as its primary input. While the curation pipeline ensures data quality, diversity, and compliance, the training pipeline transforms this data into functional language models.

The relationship with other work packages is essential to the training process. WP12 (Data & Algorithms) provides the curated training data, evaluation datasets, and algorithmic guidance for data mixing strategies. WP13 (System Architecture & Infrastructure) designs and implements the distributed training infrastructure, monitoring systems, and deployment mechanisms; and provides this architectural documentation.

This document gives an overview of the training pipeline for the GPT-NL foundation model. It gives a high-level description of the processes involved in training this model that have implications for the system architecture design and choices for the supporting software stack. *Note that document is not meant as a complete and standalone architecture design.*

The training pipeline architecture is built on three core principles that guide design decisions and implementation choices:

Reproducibility ensures every aspect of the training process can be recreated. All hyperparameters, data mixture ratios, and training settings are stored in configuration files. Data snapshots and metadata track which data was used at each training stage. Experiment tracking tools like Weights & Biases log metrics, hyperparameters. Artifacts are stored throughout running processes on the Snellius HPC. All training code is version-controlled in Git repositories.

Scalability allows the architecture to support training models of varying sizes across different scales. Dedicated libraries enable scalable training across multiple nodes using various strategies. SLURM provides job scheduling and resource management. Recovery from hardware failures or job time limits is enabled through checkpointing and restart capabilities. Performance optimizations include gradient accumulation, mixed-precision training, and activation checkpointing.

Observability provides comprehensive monitoring to ensure training health and progress. Training metrics include loss, perplexity, gradient norms, and learning rates. System metrics track GPU utilization, memory usage, communication overhead, and throughput. Periodic validation loss and benchmark performance are captured as evaluation metrics. Job lifecycle alerting systems notify teams to act quickly when needed.

1.1.1 Infrastructure & Software

The training pipeline runs on SURF's Snellius supercomputer, which provides NVIDIA H100 GPUs with NVLink and InfiniBand interconnects for high-bandwidth communication. SLURM manages job scheduling, resource allocation, and queue management. High-performance parallel file systems enable efficient data loading and checkpoint storage. For more details on the Snellius HW used in the GPT-NL curation and training see the **GPT-NL data curation and training at SURF's HPC Snellius** appendix.

The software stack includes native PyTorch with FSDP (Fully Sharded Data Parallel) for pre-training, and HuggingFace TRL with DeepSpeed for fine-tuning. Distributed computing relies on NCCL for GPU communication. Monitoring uses Weights & Biases and custom logging infrastructure. Model serving is managed through vLLM inference engine with Hugging Face Transformers. Detailed information about the software stacks and infrastructure is provided in the appendices.

1.1.2 Overview of the Training Lifecycle

The entire training process moves through a series of linked stages. Each phase consumes outputs from previous stages and produces specific artifacts that feed into subsequent steps.

The process begins with **data preparation and tokenization**, where curated data is transformed into a format suitable for model consumption. This involves training a custom tokenizer optimized for Dutch and multilingual text, combining different data sources according to designed mixture ratios, and converting raw text into tokenized sequences stored in optimized formats. The inputs are curated datasets from the data curation pipeline, and the outputs include the trained tokenizer, data mixture configurations and tokenized datasets. This phase runs on CPU-based preprocessing jobs on Snellius compute nodes.

The **pre-training phase** trains the core foundation model from scratch using large-scale distributed training across multiple GPU nodes. The process leverages parallelization mechanisms to scale to 88 H100 GPUs. Training includes continuous evaluation on held-out data and benchmark tasks, regular checkpointing of model, optimizer, and scheduler states for fault tolerance, and real-time monitoring of loss curves, learning rates, gradient norms, and resource utilization. This phase consumes the tokenized training data, tokenizer, and model architecture configuration, producing foundation model checkpoints at multiple snapshots throughout training. Pre-training is inherently iterative, as new curated data may arrive during training and can be incorporated in subsequent cycles or continuation runs.

Following initial pre-training, **context extension** adapts the model's context window from its original training length (e.g., 4K tokens) to longer contexts (16K or 32K tokens). This involves adjusting positional encodings such as RoPE and brief additional training on long-context data. The process takes the pre-trained foundation model and long-context training data as input, producing extended-context model checkpoints.

The **instruction fine-tuning phase** adapts the foundation model for specific downstream tasks using instruction-following datasets. The process involves curating and filtering instruction datasets, training the model through supervised fine-tuning to follow instructions and generate appropriate responses, and optimizing for specific tasks like summarization, simplification, and RAG capabilities. Performance is assessed on instruction-following benchmarks throughout the process. This phase uses smaller GPU allocations (8-32 GPUs) and runs for shorter durations (hours to days) compared to pre-training. Fine-tuning experiments typically iterate to optimize hyperparameters, data mixtures, and training strategies.

Finally, **model deployment** packages trained models for production use. Models are converted to deployment-ready formats compatible with Hugging Face, containerized for reproducible deployment environments, and integrated with evaluation frameworks and application interfaces. The deployed models are served via vLLM inference engine, making them accessible via API or for local inference on inference-optimized GPU servers.

1.2 Scope of the System Architecture Work and Relation to Other Work

Besides the close work developed with **WP12, WP14, and WP18**, cybersecurity and evaluation activities are also depicted in Figure 1. These activities are out of the scope of the architecture team, but their insights and outcomes influence and are influenced by the GPT-NL architecture work. For example, **WP21 Evaluation** and the **Cybersecurity work package (WP22)** operate independently to ensure objective assessment and verification. WP21 evaluates the trained model's performance on key tasks, while the cybersecurity and red-teaming teams assess its resilience and safety. WP22 is involved with securing the development and model overall (in Figure 1 depicted only at the end for readability). The team puts in place classic and AI-specific cybersecurity mechanisms. Although separate, these teams collaborate closely with WP13 by consuming its architectural artifacts, interfaces, and documentation, and by providing feedback that informs subsequent development cycles.

1.2.1 Architecture Team

The GPT-NL architecture team has a multidisciplinary composition with SW architects and engineers, open-source specialists, high performance computers architects, ML engineers, and data scientists. Members of TNO and SURF form the team. Acknowledgements for the support of SURF in all the management and proper usage of the Snellius supercomputer.

1.3 How to further read this document

This document focusses on the training of the GPT-NL foundation model, and the following sections provide detailed documentation for each phase of the training pipeline. The structure progressively increases in technical depth, from high-level architectural decisions to implementation details and code-level documentation.

The **pre-training deep dive** begins with a comprehensive overview of the pre-training process, followed by detailed sections on tokenizer fitting & tokenization, data preparation & mixing, running & monitoring pre-training, scaling strategies, context extension, model & hyperparameters, and code organization.

The **instruction fine-tuning deep dive** starts with a comprehensive overview of the fine-tuning process, followed by sections on data preparation, data selection methods, running & monitoring fine-tuning, evaluation and initial results, and code organization.

Additional sections cover model deployment for packaging and deploying trained models, and training appendices with additional technical details, software stack analysis, and framework experiments.

2 Architecture of the Pre-Training pipeline

The GPT-NL pre-training pipeline is the most compute-intensive stage of model development, transforming curated multilingual datasets into large-scale base language models for Dutch and English. This process involves the following key steps: tokenizer fitting, data preparation, distributed pre-training across high-performance compute clusters, and optional context extension for longer input sequences. The pipeline ensures that models are optimized for quality, scalability, and adaptability before fine-tuning.

Between June and December 2025, GPT-NL underwent three major training epochs and a final annealing phase, processing 1.9 trillion tokens across multiple languages and code. The resulting model—a 26B-parameter architecture based on Gemma 3—was trained using 220 H100 nodes on Snellius. This foundation enables robust multilingual capabilities and serves as the basis for subsequent fine-tuning and evaluation stages.

The full pipeline, including its main functional phases, is illustrated in Figure 3 below. At its core is the pre-training process, which optimizes a randomly initialized model to one that fits the pre-training distribution.

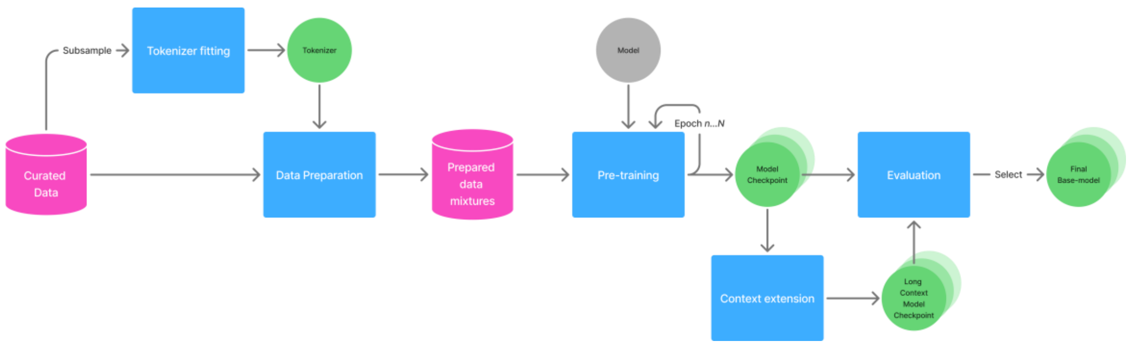


Figure 3: Overview of the pre-training pipeline

The pipeline consists of the following steps:

Tokenizer fitting - Optimizes an algorithm to efficiently encode raw text into a numerical representation that can be processed by the model.

Data preparation - Transforms the raw, curated data into tokenized form and mixed to match specific language types and quality target distributions.

Pre-training - Runs distributed training across the full compute pool (202 H100 nodes on Snellius, each consisting of 4 H100 GPUs).

Context extension - Optional phase to adapt the model to longer input sequences via longer-form content.

Evaluation - Evaluates base model capabilities across tasks to monitor training and select final artifacts.

The pre-training process is an iterative, multiphase process where every phase (epoch) processes a mixed version of the data (see Figure 4 below). When new data arrives, it can be incorporated into the pre-training from the next epoch onwards. To finalize a model, it needs to undergo a shorter annealing phase (where the data mixture is biased towards higher quality data), which can be instantiated after each intermediate epoch, leading to *n* intermediate base models.

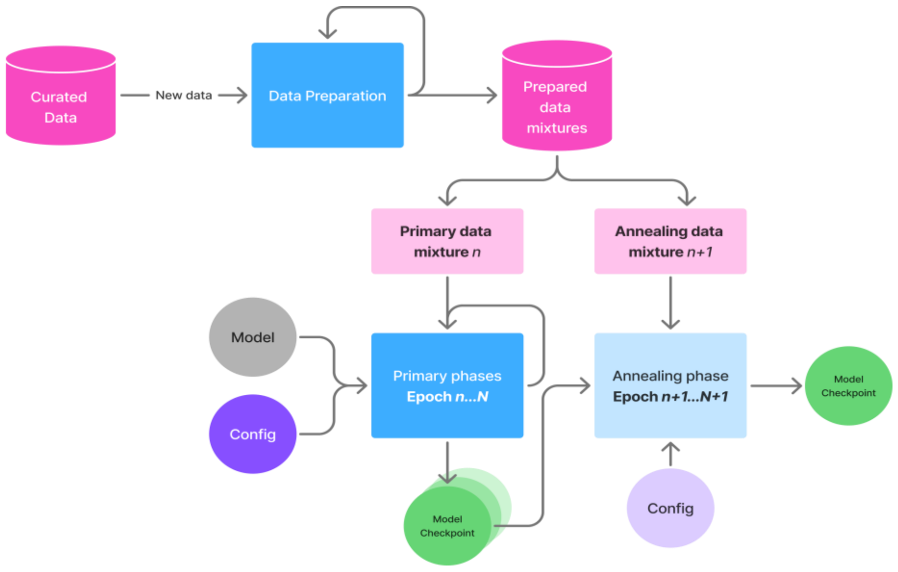


Figure 4: Depiction of multi-stage pre-training

All functionality concerning pre-training is implemented across three GitLab repositories:

1. The **Dataloader repository** that implements all steps preceding training, including training the tokenizer and tokenizing all curated text datasets.
2. The **Pytorch Native repository** that implements the pre-training process and hosts additional functionality as well, such as evaluation and running inference with the model.
3. The **Context Lengthening repository** which is solely concerned with the context extension phase.

This section details the GPT-NL pre-training as it happened between June 16th, 2025, and December 31st, 2025. Training was intermittent due to maintenance and shared infrastructure, lengthening the overall duration. The pre-training consisted of three main epochs (i.e., passes over the data) and an annealing phase. In the second epoch we added new data (mostly Dutch text) that arrived while epoch 1 had already started. The annealing phase is a short final phase emphasizing higher quality data. The total number of tokens seen by the model is 1.9 trillion tokens.

Total training tokens after up sampling (in billions), seen throughout training:

Language	Epoch 1	Epoch 2	Epoch 3	Annealing
English	279.78	269.10	269.10	266.24
Dutch	139.89	116.74	116.74	73.20
Code	83.93	80.73	80.73	31.71
Other languages	55.96	53.82	53.82	7.93
Total	559.56	520.40	520.40	379.07
Cumulative total	559.56	1079.96	1600.36	1979.44

The trained model has 26B parameters and is based on the Gemma 3 architecture. For more details on the architecture, see the training hyperparameters Section.

The following subsections dive into each of the key steps of the pre-training.

2.1 GPT-NL Tokenizer

A tokenizer is an algorithm that converts raw text into numerical tokens, enabling a language model to process the data effectively. For GPT-NL, a custom tokenizer is trained from scratch to ensure efficient representation of Dutch text, which is often underrepresented in existing multilingual tokenizers.

Tokenization refers to the process of transforming a text dataset into this numerical form. A trained tokenizer segments text into chunks based on their statistical occurrence and maps these segments to numerical representations. Tokens may correspond to individual characters, sub-word units, or frequently occurring combinations of characters and symbols.

The figure below illustrates tokenization by depicting tokens within a sentence, where each token is represented by a distinct colour.

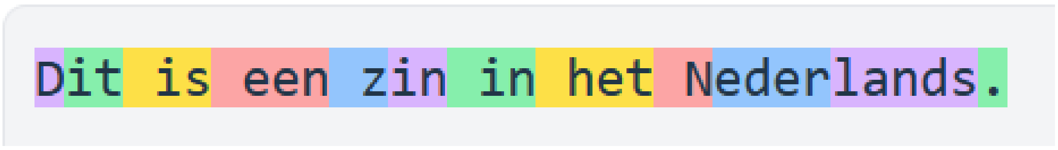


Figure 5: A tokenized sentence. Tokens can represent single letters, parts of a word, or frequent sequences of letters and symbols.

The entire tokenization involves roughly 2 stages:

- Training the tokenizer from scratch
- Tokenizing the GPT-NL data in preparation for the model training

2.1.1 Tokenizer training

Training a tokenizer at scale involves processing a high amount of data (in the lines of 100+ GB of raw text). Frameworks like Hugging Face's tokenizers are not capable of efficiently tokenizing the data even with 1TB+ of RAM. In contrast, SentencePiece does work reasonable with high volumes of data, hence the GPT-NL tokenizer was trained with the SentencePiece library.

First, a subset of the training data is sampled to obtain the tokenizer training set. To be precise, 120GB of raw text was employed for the tokenizer set with the following language mixture:

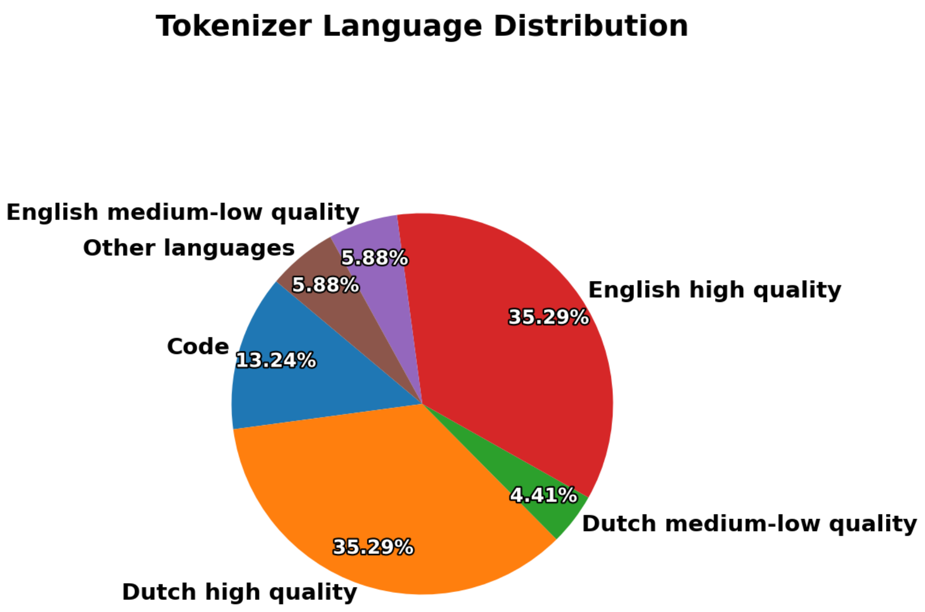


Figure 6: Tokenizer training language distribution

Secondly, the SentencePiece tokenizer training configuration was set. This includes initializing the byte-fallback Byte-Pair Encoding tokenizer, setting the final vocabulary size of 128k, special tokens definition and other options. The tokenizer in fact is similar to the Llama tokenizer and follows the HuggingFace's LlamaTokenizer class. The configuration of SentencePiece can be found in the internal GitLab here.

Training the tokenizer took 64 hours on a `fat_genoa` node with 192 cores and 1440 GiB of memory.

Lastly, the tokenizer is converted from the SentencePiece format to the HuggingFace format such that it can be loaded with

```
from transformers import AutoTokenizer
gptnl_tokenizer = AutoTokenizer.from_pretrained(<path_to_tokenizer>)
```

The GPT-NL tokenizer is initialized with 100 reserved tokens for downstream adjustments and in the form of `<reserved_token_xx>`. As the tokenizer follows LlamaTokenizer closely, the following default tokens are considered:

```
{
      "id": 0,
      "content": "<unk>",
      "single_word": false,
      "lstrip": false,
      "rstrip": false,
      "normalized": false,
      "special": true
    },
    {
      "id": 1,
      "content": "<s>",
      "single_word": false,
      "lstrip": false,
      "rstrip": false,
      "normalized": false,
      "special": true
    },
    {
      "id": 2,
      "content": "</s>",
      "single_word": false,
      "lstrip": false,
      "rstrip": false,
      "normalized": false,
      "special": true
    },
    {
      "id": 3,
      "content": "<pad>",
      "single_word": false,
      "lstrip": false,
      "rstrip": false,
      "normalized": false,
      "special": true
    },
```

For instruction-tuned tokenizers, we follow the ChatML templates:

```
"4": {
      "content": "<|im_start|>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "5": {
      "content": "<|im_end|>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "6": {
      "content": "<|system|>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "7": {
      "content": "<|user|>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
    "8": {
      "content": "<|assistant|>",
      "lstrip": false,
      "normalized": false,
      "rstrip": false,
      "single_word": false,
      "special": true
    },
```

2.1.2 Tokenization

The OLMo-core pretraining pipeline expects tokenized `.npy` (NumPy memory-mapped files) as its training data. To this end, the entire dataset is tokenized before training. Specifically, the data is put into smaller buckets of file and by employing the SLURM job array, tokenizing the entire dataset with parallel jobs only took a couple of hours.

Tokenizer fertility rate is defined as the average number of tokens produced per word in a given text. A lower fertility rate is generally preferable because it indicates that words are represented more compactly, which reduces sequence length and enhances model efficiency. When fertility is high, token sequences become longer, leading to increased memory consumption and slower inference times.

Optimizing tokenizers for lower fertility ensures that models process text more efficiently without sacrificing semantic integrity. By reducing the number of tokens per word, the computational workload decreases, directly lowering training and inference costs and making large-scale language model operations more economical.

Below is the tokenizer fertility rate compared to competitive multilingual tokenizers. The performance is reported on a random subset of Dutch data:

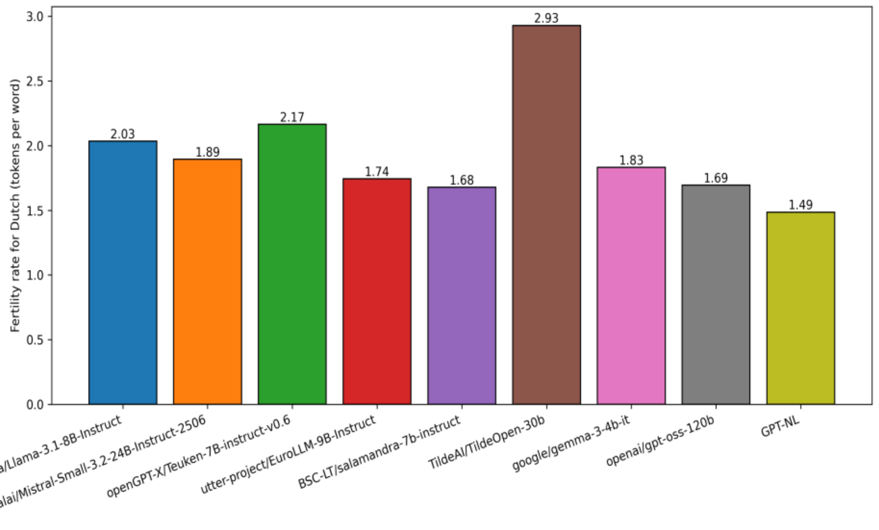


Figure 7: Tokenizer comparison (lower means a better fit)

Given the different vocabulary sizes of 128k for GPT-NL and the competitive Salamandra of 256k, thereby adding extra parameters in the embedding space, the GPT-NL tokenizer shows a good trade-off between fertility/token efficiency and model complexity.

Numeric evaluations of the tokenizer are discussed in the Evaluations Section and in the Data Preparation Section.

2.2 Pre-training Data Preparation

This section describes the complete process of transforming curated datasets into optimized data mixtures ready for pre-training. The data preparation pipeline connects the data curation phase (delivered by WP12) with the pre-training phase, producing data mixtures that support Dutch language representation and allow for quality control of input sources.

The pipeline is implemented in the **Dataloader GitLab project** and encompasses tokenizer fitting, data bucketing, train-validation splitting, tokenization, and distribution mixing. The process is guided by two key variables: **detected language** and **quality assessment**, which together define the characteristics of different data buckets used to construct training mixtures for different phases of pre-training.

2.2.1 Motivation and Design Principles

One primary objective of GPT-NL is to achieve adequate representation of the Dutch language in the final model. The Dutch share of available data is low relative to English, even after receiving additional Dutch data during epoch 1 training, the size remains disproportionate compared to English resources. To address this imbalance, we up sample the Dutch data share during training.

Data quality also affects model performance. Recent large-scale model reports provide evidence for quality-focused data strategies. The Llama 3 team (Llama AI Team, 2024) found that using high-quality code and mathematical data in the final training phase can boost performance on key benchmarks, with experiments showing improvements (24% on GSM8k and 6.4% on MATH) for smaller models. The benefits were less pronounced for larger models that already exhibited stronger reasoning capabilities.

Similarly, EuroLLM (Martins et al., 2024)—a multilingual European language model project—reports a comparable strategy: in the last 10% of pre-training, they increase the presence of high-quality data in the mix. EuroLLM filters monolingual data using a binary classifier inspired by FineWeb-Edu (Penedo et al., 2024) to predict whether documents have educational value, and incorporates additional high-quality datasets including Cosmopedia-v2, Python-Edu, training sets from GSM8K and MATH benchmarks, and document-level parallel data from Europarl and ParaDocs.

Following these approaches, GPT-NL employs a **bucket system** to control both language distribution and data quality throughout pre-training. We categorize all curated data into buckets based on two primary dimensions:

1. **Language**: Detected language (Dutch, English, Code, Other languages)
2. **Quality**: Manually attributed quality assessment labels provided by WP18 at dataset level (high, medium, low)

This bucketing strategy enables us to create specific data mixtures for different training phases—maintaining broader diversity during the primary training epochs (1-3) while biasing toward higher quality data during the final training phase (i.e., the **annealing phase**). The system also tracks newly added data separately, allowing us to incorporate fresh content that arrived during training into subsequent epochs.

The following sections detail the quality assessment methodology, provide concrete examples of bucket characteristics, describe the data preparation pipeline implementation, and present the final data mixtures used throughout GPT-NL pre-training.

2.2.2 Quality Assessment and Bucketing Methodology

Quality assessment for GPT-NL datasets was performed manually by WP18 at the dataset level. Rather than applying strict, quantitative criteria, the assessment considers the nature and characteristics of each data source to assign quality labels (high, medium, or low). The table below illustrates representative examples across the quality spectrum.

Source	Quality	Rationale
OpenRaadsInformatie — Public hearings and decision documents from ~350 Dutch municipalities, water boards, and provinces	High	Professionally written transcripts with high expected accuracy
NDP National-Regional — National and regional newspapers from members of the Dutch news branch organization	High	Content written by professional journalists and published in established newspapers
Fryske Akademy — Content from the Frisian language and culture research institute	High	Curated Frisian language content from an academic institution
Synthetic Wikidata — Synthetically generated content based on structured Wikidata entries	Medium	Systematic generation from structured high quality data, but generation process may introduce errors
KPN Web Content — Web content from the Dutch telecommunications company	Medium	Web content that may contain errors or inaccuracies typical of online sources
CommonCrawl Creative Commons — Dutch CC-BY and public domain web content from CommonCrawl	Medium	Variable quality typical of web scraping, though filtered for permissive licenses
Noord-Hollands Archief — Archives older than 100 years from the provincial archive of North Holland	Low	Contains numerous OCR errors from digitization of historical documents
YouTube Commons — CC-BY video transcripts from YouTube	Low	Automatic transcripts that are often inaccurate
Nationaal Archief — Digitized historical archives including VOC records	Low	Digitization error rate of approximately 8%, resulting in substantial OCR errors, particularly in the VOC subset

Quality assessment is based on multiple criteria including the reputation and origin of the source, recency of the content, and technical quality factors such as OCR accuracy.

High-quality sources typically originate from institutions or organizations with editorial standards or quality control processes. These include professional journalism, official government documentation, and curated academic content. The writing in these sources is produced with attention to accuracy and coherence. Sources in this category have strong reputations and provide reliable, well-structured content.

Medium-quality sources encompass web content and synthetically generated material. While web content from reputable organizations or filtered sources may be generally reliable, it lacks the editorial oversight of high-quality sources. Synthetic content generated from structured data sources like Wikidata is systematic but may contain artifacts from the generation process.

Low-quality sources primarily suffer from technical limitations in data capture or transcription. Automatic transcription systems and OCR processes applied to historical documents introduce errors that degrade text quality. These sources remain valuable for their content and linguistic diversity despite their technical imperfections.

The quality labels are applied at the dataset level rather than at the document level, meaning all documents from a given source receive the same quality designation. For the data preparation pipeline, low and medium quality data are combined into single buckets. This simplification is sufficient because quality distinction is only applied during the final, short training phase (annealing), where we bias toward high-quality data. For this purpose, distinguishing high-quality sources from all others provides adequate granularity without requiring a large volume of finely categorized data.

Beyond quality labels, the bucketing process relies on additional metadata fields to categorize data. Language detection is performed using automated language identification tools as described in the curation stages. Code datasets are identified through manual labelling, with one primary code dataset comprising the majority of code content. Temporal metadata tracks when data arrived—either before training began or during epoch 1, allowing newly added data to be incorporated into subsequent epochs while maintaining separate tracking for mixture composition purposes.

2.2.3 Data Bucket Characteristics and Examples

To illustrate the differences between buckets, this section provides representative text samples from each category. These examples demonstrate the characteristics that distinguish high-quality sources from lower-quality ones, and show the linguistic diversity across Dutch, English, code, and other language buckets. Each sample is extracted from actual datasets used in training.

2.2.3.1 English High Quality

Dataset: Common Corpus

Text:

```
ABSTRACT The palm (Phoenix Dactylifera) is one of important trees and is ec
onomically important in south of Iran. Date palm is propagated by the offsh
oots, number of which is limited. Therefore, adul...<truncated>...eijer., &
Levi van de Biezenbos. (1993). Occurrence of direct somatic embryogenesis o
n the sword leaf of in vitro plantlets of Phoenix dactylifera L. cultivar b
arhee. Current Science, 887-889. 430 430
```

2.2.3.2 English Low/Medium Quality

Dataset: Common Corpus

Text:

```
The Evangelicals, who had been quickened to seek the spread of the Gospel a
broad through the institution, in 1822, of " La Soci6t6 des Missions Evangd
liques chez les Peuples non-Chr6tiens," earnestly ...<truncated>...tab-' li
shments, supervised by 7000 nuns and served by 48,000 women, bringing an an
nual income, from the unpaid labour of the pensionnaires, of not less than
;^6oo,ooo.^ The Associations Bill of 1901.
```

2.2.3.3 Dutch High Quality

Dataset: Open Raadsinformatie

Text:

```
**Bouwsteen Economie** Voor deze bouwsteen economie is een rapport opgestel
d door een extern bureau. Dit rapport dient als aanbeveling aan gemeente Zw
olle. Het rapport doet aanbevelingen over de toeko...<truncated>... van de
stad (voor de verschillende opleidingsniveaus)| **Overi** **g** **e o** **p
** **merkin** **g** **en** **,** **ideeën etc.** Kun je iets niet kwijt ond
er bovenstaande? Daar is hier plek voor. 7
```

2.2.3.4 Dutch Low/Medium Quality

Dataset: Woogle

Text:

```
Toetsing Op grond van artikel 5.16 lid 1 van de Wet milieubeheer kan de ver
gunning alleen worden verleend, als aannemelijk gemaakt kan worden dat vold
aan wordt aan (minimaal) één van de volgende crite...<truncated>...rgunning
houder de resultaten daarvan wil implementeren, daartoe eerst steeds zal mo
etel worden bezien in hoeverre een procedure op grond van de Wabo zal moete
n worden doorlopên. Zaaknum m er: 78891724
```

2.2.3.5 New (epoch 2+) Dutch High Quality

Dataset: NDP

Text:

```
'Chris' wil prijs pakken in Los Angeles CALL OF DUTY Silvano heeft talent v
oor Call of Duty. Hij is zo goed dat hij volgende week gratis naar Los Ange
les mag. door Dewi Willems van Lier MIDDELBURG -Vi...<truncated>...rond te
kijken. Als ik naar het strand wil, brengen ze me erheen. Ik heb vorige kee
r een hoop Ferrari's gezien. Die wil ik deze keer wat beter bekijken. " fot
o Isabella Oosterhek-Booden Puk Langevoort
```

2.2.3.6 New (epoch 2+) Dutch Low/Medium Quality

Dataset: YouTube-Commons

Text:

```
Hey jongens welkom terug bij een andere video bitcoin is blijven dumpen lat
en we eens kijken naar de grootste winnaars en verliezers in de altcoin-rui
mte het gaat om een soort van top 10 top 20 altcoi...<truncated>...is, is d
e link naar de investeerdersaccelerator in de beschrijving hieronder, we zi
en je daar in de 12-maanden lidmaatschapsgroep, maar tot de volgende keer,
heb meer plezier om meer gedaan te krijgen
```

2.2.3.7 Other Languages (all quality levels)

Dataset: Common Corpus

Text:

```
doppelten , diese ziehen sich des Wund- winkels h (in der Rich- tung nach d
) ein Ectropium zu Stande kommen. Tritt aber nun noch der Fall ein, dass di
e prima in- tentio nicht erfolgt, dass durch Eiter...<truncated>... die Wis
```

```
senschaft dieselben weglassen, da eine geübte Hand mit einer gewöhnli- chen
Hakenpincette und gerader oder gebogener Scheere die Falte ebenso gut den V
erhältnissen entsprechend entfernen kann.
```

2.2.3.8 Code

Dataset: Common Corpus

Text:

```
package com.foo.bar.steps import com.foo.bar.SmsVerificationCodeSenderStub
import com.thebund1st.daming.commands.SendSmsVerificationCodeCommand import
com.thebund1st.daming.core.SmsVerificationCode im...<truncated>...()) .body
("token", notNullValue()) } def then(String description) { this } def theCo
deReceived() { code } def shouldSeeFailure(HttpStatus httpStatus) { this.re
sponse.statusCode(httpStatus.value()) } }
```

2.2.4 Data Availability Across Training Epochs

New data arrived throughout training, changing the composition and scale of available data between epochs. The table below summarizes the token counts in billions available during epoch 1 and from epoch 2 onwards (denoted as Epoch 2+), with percentages relative to each epoch's total. These counts include both training and validation data.

Bucket Category	Epoch 1 (B)	Epoch 1 %	Epoch 2+ (B)	Epoch 2+ %
Other languages	48.33	13.31	48.34	8.80
English (low + medium)	158.39	43.63	158.39	28.84
English (high quality)	49.52	13.64	49.52	9.02
Dutch (low + medium)	6.26	1.72	14.76	2.69
Dutch (high quality)	14.88	4.10	46.24	8.42
Code	85.64	23.59	231.95	42.23
Total	**363.01**	**100**	**549.20**	**100**

The data landscape changed substantially between epochs. Most notably, the Code bucket increased from 85.64B to 231.95B tokens (a 2.7× increase) due to an adjustment in the filtering steps of the data curation, shifting its percentage from 23.59% to 42.23% of the total. Dutch high-quality data increased significantly from 14.88B to 46.24B tokens (a 3.1× increase), more than tripling its representation. This influx of Dutch content reflects additional data that arrived during epoch 1 training. The relative percentages of other categories decreased correspondingly due to the overall growth in total available data from 363B to 549B tokens.

2.2.5 Data Preparation Pipeline

The data preparation pipeline implementation transforms curated datasets into training-ready data mixtures through a multi-stage process. The complete pipeline is visualized in the diagram below, showing the flow from input parquet files through bucketing, train-validation splitting, tokenization, and distribution mixing phases.

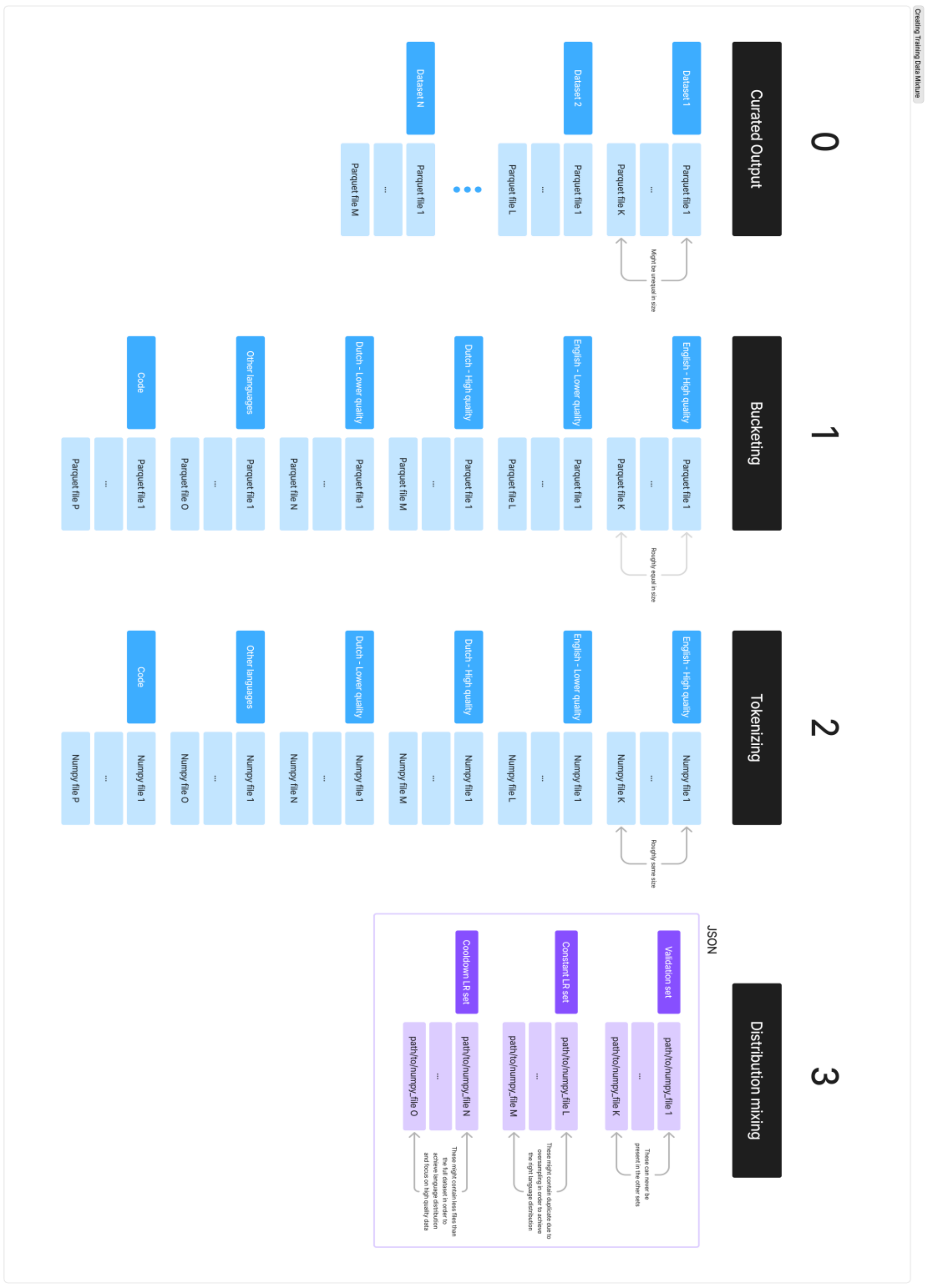


Figure 8: Diagram multi-stage process of creating the pre-training data mixture

The pipeline takes as input files in parquet format organized by data provider or openly available dataset and processes them through four main phases: 1) bucketing, 2) train-validation split, 3) tokenization, and 4) distribution mixing to create optimized data mixtures for model training.

The pipeline begins with curated datasets (as delivered by WP12) stored in parquet file format, organized in subdirectories by dataset name (from a GPT-NL data provider or open dataset). Each dataset has slightly different schemas due to varying application of curation stages (e.g., Personal Identifiable Information (PII) detection might not be applied to all sets), but files within the same dataset maintain consistent structure. Files vary in size due to filtering steps employed during curation (disregarding documents, or rows, from the parquet files).

2.2.5.1 Phase 1: Bucketing

The bucketing phase groups data by language, quality attributes, and whether the data is newly added in that epoch, organizing it into structured containers. These buckets represent key characteristics that define our desired training mixtures for both the primary and annealing phases of pre-training.

Metadata Investigation & Assembly

This stage analyses available curated datasets to understand their structure and content, and leverages metadata provided by human annotators (WP18) to further tag the data with quality labels (high/medium/low) and stores metadata about which datasets contain code (as described in the Quality Assessment section).

Data Tagging

This stage enriches existing parquet files with additional metadata columns, adding standardized information including:

- A unique document ID
- Original parquet file path as delivered by WP12
- Original dataset name and quality assessment
- Language detection results
- Code dataset classification flags

Data Bucketing

This stage distributes data into eight primary buckets based on language, quality, and time of arrival (before or during training):

3. New data high quality (only relevant for epochs > 1, assumed to be almost all Dutch)
4. New data low + medium (only relevant for epochs > 1, assumed to be almost all Dutch)
5. English high quality
6. English low + medium quality
7. Dutch high quality
8. Dutch low + medium quality
9. Other languages
10. Code

This stage is implemented by first **sorting** the data into temporary files and then **merging** them to have equal size (in number of rows or in file size).

2.2.5.2 Phase 2: Train-Validation Split

This phase implements a stratified train-validation split for bucketed parquet datasets. It takes categorized data files (organized in buckets like `dutch_high_quality`, `english_low_medium_quality`, `code`, etc.) and splits each file into training and validation sets while maintaining proportional representation of all source datasets.

The split uses a **98-to-2 ratio** (keeping validation small to maximize training data) and stratifies by dataset folder (extracted from the `original_file_path` column) to ensure validation sets contain samples from all original datasets. The process handles edge cases where some datasets have insufficient samples for stratification by placing single-sample categories into training, and processes files in parallel for efficiency while managing memory usage through garbage collection and process pool controls.

This split is necessary to evaluate model performance on held-out data during training, which helps prevent overfitting, monitor training progress through validation loss, and ensure the validation set is representative of all source datasets. While this is standard practice in machine learning, maintaining this separation is important for transparent evaluation of training progress.

2.2.5.3 Phase 3: Tokenization

This phase converts text data into token sequences suitable for model training, stored in memory-mapped NumPy files (memory-mapped for efficient reading during training).

The tokenization process uses the fitted tokenizer (output of the process described in Section 2.1) to:

- Tokenize all bucketed data using the trained tokenizer
- Implement sequence packing strategies for training efficiency
- Output tokenized files organized by bucket for downstream processing

2.2.5.4 Phase 4: Distribution Mixing (Sampling)

This phase creates training data mixtures optimized for different training phases. A sampling module enables flexible mixture construction with the following capabilities:

- Sample a mixture with a desired language distribution by over- or under sampling files to achieve the target distribution
- Bias sampling toward higher quality files to achieve a higher quality data mixture
- Constrain the mixture to a specific dataset size

With this module, the following data mixtures are created for different training phases:

- For epochs in the **primary training phase**: Sets sampled with a desired language distribution with **no bias** for higher quality data
- For the **annealing phase**: A set sampled with a desired language distribution **with bias** for higher quality data, constrained to consist of about 15% of the primary set in size

Sampling happens at file level, with files containing approximately 1GB worth of documents.

The sampling process generates structured file lists as output:

- Primary phase file paths for main training (train and validation) (for epochs 1, 2, and 3)
- Annealing phase file paths for final training refinement

2.2.6 Target Data Mixtures

For both pre-training phases (primary and annealing) we set a desired language distribution to balance the goals of GPT-NL with the available data. As the annealing phase is typically short (10-15% of pre-training), less upsampling is required to achieve the target distribution. In this phase we bias toward higher quality data by only selecting from high-quality subcategories.

For the primary phase (epochs 1-3), we target the following language distribution:

Language	Target Percentage
Dutch	25%
English	50%
Code	15%
Other	10%

These percentages balance GPT-NL's objective of adequate Dutch language representation with the available data while maintaining multilingual capabilities through English and other languages and incorporating substantial code data to support technical understanding and reasoning capabilities.

For the annealing phase, we target a distribution more pronounced toward the Dutch language:

Language	Target Percentage
Dutch	35%
English	40%
Code	20%
Other	5%

The annealing phase constitutes approximately 10-15% of total pre-training. The shift toward more Dutch content (from 25% to 35%) and increased code representation (from 15% to 20%) reflects the strategy of emphasizing these areas in the final training stage. For this phase, only high-quality data buckets are used, excluding all low and medium quality sources.

To achieve the target distributions with the data available in epoch 1 and in epochs 2 and 3, we applied different upsampling rates to each bucket:

	Language	Epoch 1 Tokens (B)	Epoch 1 Upsamp. Rate	Epoch 1 %	Epoch 2 & 3 Tokens (B)	Epoch 2 & 3 Upsamp. Rate	Epoch 2 & 3 %	Annealing Tokens (B)	Annealing Upsamp. Rate	Annealing %
Code	code	83.93	1	15	80.73	0.36	15.51	80.73	0.36	15.51
English low quality	en	213.14	1.37	38.09	205.01	1.32	39.39	205.01	1.32	39.39
English high quality	en	66.64	1.37	11.91	64.09	1.32	12.32	64.09	1.32	12.32
en_subtotal	en	279.78	2.75	50	269.1	2.64	51.71	269.1	2.64	51.71
Dutch low quality	nl	41.41	6.75	7.4	6.13	1	1.18	6.13	1	1.18

Dutch high quality	nl	98.48	6.75	17.6	20.88	1	4.01	20.88	1	4.01
New Dutch low quality	nl	-	-	-	22.81	2.74	4.38	22.81	2.74	4.38
New Dutch high quality	nl	-	-	-	66.92	2.74	12.86	66.92	2.74	12.86
nl_sub-total	nl	139.89	13.5	25	116.74	7.48	22.43	116.74	7.48	22.43
Other languages	other	55.96	1.18	10	53.82	1.14	10.34	53.82	1.14	10.34
total		559.56	18.43	100	520.4	11.61	100	520.4	11.61	100

The upsampling rates were calculated to meet target distributions given available data in each bucket. Dutch data required the highest upsampling rates (6.75× in epoch 1, decreasing to 2.74× for new Dutch data in epochs 2-3) to achieve the 25% target in primary training. English data required modest upsampling (1.37× and 1.32×), while Code data was actually downsampled in epochs 2-3 (0.36×) due to the large influx of code content. The resulting mixtures closely approximate the target distributions while incorporating all available data according to the defined sampling strategy.

2.2.7 Pipeline Output and Folder Structure

The data preparation pipeline produces outputs at each stage, organized in a structured folder hierarchy on the Snellius HPC system. The pipeline begins with curated datasets and progresses through bucketing, train-validation splitting, tokenization, and sampling phases, with each phase producing organized outputs consumed by subsequent stages. Data is stored on the project data share.

Location: `/projects/0/prjs0986/wp12/dataset_delivery/`

```
wp12/dataset_delivery/
├── gpt_nl_dataset_v1.0/   # Parquet files organized by dataset provider
│   ├── american-stories/
│   ├── cc_english-pd/
│   ├── kb/
│   └── ... (additional datasets)
└── gpt_nl_dataset_v2.0/ # Same structure as v1.0, available from epoch 2
```

Raw curated datasets in parquet format are organized by dataset provider or open-source dataset name.

2.2.7.1 Phase 1 Output: Bucketing

Location: `/projects/0/prjs0986/wp14/<gpt_nl_data_version>_bucketed/`

After tagging and bucketing, data is organized into eight buckets:

```
bucketing_output/
├── dutch_high_quality/
├── dutch_low_medium_quality/
├── english_high_quality/
├── english_low_medium_quality/
├── other_languages/
├── code/
├── new_high_quality/          # (epoch > 1 only)
└── new_low_medium_quality/    # (epoch > 1 only)
```

Each folder contains parquet files of similar size (approximately 1GB) with enriched metadata including document IDs, quality labels, language tags, and original file paths.

2.2.7.2 Phase 2 Output: Train-Validation Split

Location: `/projects/0/prjs0986/wp14/<gpt_nl_data_version>_bucketed/`

Data is split into training (98%) and validation (2%) sets while maintaining the bucket structure:

```
├── buckets_split_train/        # Training files by bucket
└── buckets_split_validate/     # Validation files by bucket
```

The stratified split ensures validation sets contain representative samples from all source datasets within each bucket.

2.2.7.3 Phase 3 Output: Tokenization

Location: `/projects/0/prjs0986/wp14/<gpt_nl_data_version>_tokenized/`

Tokenized data is stored as memory-mapped NumPy arrays with uint32 dtype:

```
├── buckets_split_train/        # .npy files by bucket (training)
├── buckets_split_validate/     # .npy files by bucket (validation)
└── splitted_buckets_{train,validate}/ # Reorganized structure for sampling
```

Memory-mapped files enable efficient reading during training without loading entire datasets into memory.

2.2.7.4 Phase 4 Output: Distribution Mixing (Sampling)

Location:`/projects/0/prjs0986/wp14/<gpt_nl_data_version>_tokenized/train_datamixtures/`

The sampling phase produces JSON files specifying data mixtures consumed by the training process:

- **`validation_files.json`** - File paths stratified across all buckets (approximately 2% of total data)
- **`primary_phase_files.json`** - File paths for epochs 1-3 with balanced language distribution and no quality bias
- **`annealing_phase_files.json`** - Quality-biased subset (approximately 15% of primary phase size) using only high-quality buckets
- **`mixture_statistics.json`** - Metadata including file counts, token counts, and distributions per phase

Each JSON file organizes file paths by bucket, enabling efficient parallel data loading during distributed training. The training infrastructure consumes these file lists to construct data loaders that sample from the specified files according to the upsampling rates defined for each training phase.

2.2.8 References

1. Llama AI Team (2024) | The Llama 3 Herd of Models
2. Martins et al. (2024) | EuroLLM: Multilingual Language Models for Europe
3. Penedo et al. (2024) | The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale

2.3 Pre-training model & hyperparameters

Developing a large language model requires making informed decisions regarding both model architecture and hyperparameter configuration. Architectural choices include the number of layers, the specific transformer design, and the structure of embeddings. In addition, selecting appropriate hyperparameters—such as the total number of parameters, learning rate, and optimization strategy—is critical to achieving robust performance and efficient training.

The GPT-NL model is based on the Llama 3 architecture, because of its proven performance, adaptability and strong support in modern training frameworks, making it a robust and well-understood basis for large-scale model development. It also aligned well with our choice of pre-training codebase as the OLMo model architecture is also similar to the Llama modelarchitecture.

At its core, the architecture is a decoder-only transformer architecture based on Vaswani et al. (2017). Building on this, the architecture adopts several improvements that have become standard in modern LLMs, including:

- RoPE embeddings: to allow for longer context scaling. For information on how this context is lengthened after pre-training, see this page.
- Grouped query attention: by reducing the number of key-value heads, the memory and computing requirements are reduced with minimal impact on model quality.
- SwiGLU activations: more efficient and stable than ReLU and GeLU.

2.3.1 Hyperparameters

2.3.1.1 Size

The model size is represented in the number of parameters, which depends on architectural choices like dimension sizes and number of layers. To decide on this size, we tried to find an optimal balance between model performance and computational feasibility. While smaller models (around 7B parameters) were deemed insufficient for our capability's requirements, larger models (70B+) would be too resource-intensive for our current infrastructure. The 26B parameter size should adequately handle critical tasks like summarization and retrieval-augmented generation (RAG) while still allowing for efficient inference and fine-tuning.

From a compute-optimal perspective, traditional Chinchilla-style scaling laws (Hoffmann et al., 2022) would suggest using more parameters (around 40B) for the available compute. However, these scaling laws assume training for a single epochoptimize for training compute only, ignoring inference compute optimization, and do not fully reflect the training set-ups of modern LLMs. We also refer to the Llama 3 technical blog which found continued improvement after training two orders of magnitude more data than Chinchilla-optimal. In practice, the resulting compute-quality trade-off supports a somewhat smaller model without significant loss in model performance.

The exact total number of parameters of the 26B GPT-NL model ends up at:

Parameter Type	Count
Total parameters	26,034,640,896
Non-embedding parameters	25,248,208,896
Trainable parameters	26,034,640,896

where non-embedding parameters exclude the input and output embedding matrices (whose sizes depend largely on vocabulary size and so on tokenizer design).

2.3.1.2 Learning rate schedule

For the pre-training we employ a trapezoidal scheduler (also known as WSD: warmup-stable-decay, described extensively by Hägele et al., 2024) that offers mostly practical benefits while being as performant as a cosine scheduler (the previous state-of-the-art standard). A cosine scheduler requires information of the full training length a priori. The trapezoidal scheduler consists of three phases:

4. A short, linear warm-up phase (e.g. of 2000 steps)
5. A constant learning rate phase (80-85% of steps) (we will refer to this as the *primary phase*)
6. A linear cool-down or decay phase (15-20% of steps) (we will refer to this as the *annealing* phase)

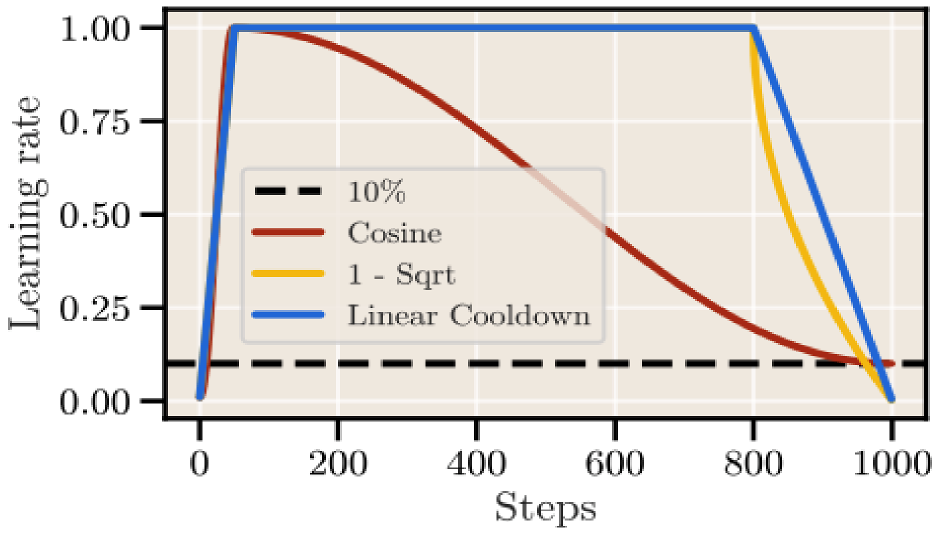


Figure 9: Illustration of cosine versus trapezoidal (blue) learning rates (source https://doi.org/10.48550/arXiv.2405.18392)

The primary advantage of the trapezoidal learning rate schedule is its flexibility during the initial phase. This phase can be extended if the model continues to improve, delaying the transition to the cool-down annealing phase. The cool-down phase stabilizes training and enables fine-grained parameter updates.

Another benefit is the ability to resume training from an earlier checkpoint within the constant learning rate phase before entering the cool-down stage. This approach is particularly efficient in experimental settings where results must be collected across varying training lengths. With a trapezoidal scheduler, these variations can be achieved using checkpoints from a single training run. In contrast, cosine scheduling requires retraining from scratch for each configuration to ensure fair comparisons and interpolation across points.

Finally, during the final training run, the trapezoidal schedule also supports an increased number of epochs, providing additional flexibility for model convergence.

2.3.2 Hyperparameter overview

The following table presents a concise overview of all the chosen GPT-NL hyperparameters:

Category	Hyper parameter	Value	Notes
Hardware			
	Number of nodes	22	During hardware failures or maintenance, the number of nodes was sometimes temporarily decreased (see also the documentation on scaling).
	Number of GPUs	88	4 x H100 GPU per node.
Architecture			
	Hidden embedding size	6144	
	Vocabulary size	128,000	See documentation on tokenizer.
	Number of layers	48	
	Number of heads	32	
	Number of key-value heads	16	
	RoPE theta	500,000	Parameter for rotary positional embeddings
	Context length	4096	See this documentation page for further information.
	Activation	SwiGLU	
Batching			
	Micro batch size	8	
	Gradient accumulation steps	3	Increases effective batch size by accumulating gradients over multiple steps.
	Global batch size	704	Number of GPUs x micro batch size
Optimizer			
	Optimizer	AdamW	
	AdamW betas	0.9, 0.95	
	Weight decay	0.1	
	Max gradient norm	1.0	
Learning rate scheduler			
	Constant learning rate	1e-4	
	Warm-up steps	2000	
	Minimum warm-up learning rate	3e-5	

	Cool-down/annealing steps	12,448	
	Minimum cool-down learning rate	3e-5	

2.3.3 References

1. Hägele et al., 2024 | Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations
2. Hoffmann et al., 2022 | Training Compute-Optimal Large Language Models
3. Llama 3 team | The Llama 3 Herd of Models

2.4 Configuring, Running, Monitoring, and Logging Pre-Training

This section describes **how a single pre-training run is orchestrated end-to-end**, from configuration to monitoring and logging. We structure the system into four tightly coupled phases:

7. **Configure (A – Training recipe)**
   Define *what* is being optimized: objective, batch geometry, optimizer, schedule, and compute kernels.

8. **Launch (B – Jobs: debug vs production)**
   Decide *where and how* the recipe runs on the HPC cluster: resource shape, provenance checks, restart policy.

9. **Monitor (C – Pre-training health)**
   Continuously track *whether training is progressing as expected*: training dynamics, throughput, and run continuity.

10. **Log (D – Persistent record & artifacts)**
    Ensure that *everything needed to replay or audit the run*—metrics, configs, checkpoints—is durably recorded.

Phases **A–D** form a loop: configuration drives launch; launch activates monitoring; monitoring writes to the logging substrate; logging feeds back into configuration and launch decisions for subsequent runs. Below we describe each phase with its purpose, technique utilized, how it works in our stack, and relevant parameters. We connect design choices to literature where appropriate.

2.4.1 Phase A: Training recipe — how we shape the optimization problem

At this phase we settle each optimization step: batch, optimizer, schedule, and compute path. Configuration is YAML-driven and supports multiple environments, so that the same semantic recipe can run in both debug and production.

Batch formation

Purpose: Control the *effective* global batch size and thus optimization dynamics, stability, and hardware utilization. Large-batch training is known to require careful learning-rate and warmup tuning to avoid degradation.

Technique: We express the global batch in *tokens*, derived from three knobs:

- `per_device_batch_size: ${oc.env:PER_DEVICE_BATCH_SIZE,12}`
- `gradient_accumulation_steps: ${oc.env:GRADIENT_ACCUMULATION_STEPS,1}`
- `num_nodes: ${oc.env:SLURM_JOB_NUM_NODES,1}`

and compute:

$$\begin{aligned}&\text{global\_tokens\_per\_update}\\ &= \text{per\_device\_batch\_size} \times \text{sequence\_length} \times \text{world\_size}\\ &\times \text{gradient\_accumulation\_steps}\end{aligned}$$

How it works in our stack:

- YAML is resolved with **OmegaConf's `oc.env` interpolation**, so we can change `PER_DEVICE_BATCH_SIZE`, `GRADIENT_ACCUMULATION_STEPS`, or node count from SLURM, without editing the config file itself.
- `global_batch_size` in `NumpyDataLoaderConfig` is set in *tokens*; the data loader handles packing that many tokens per update.
- This keeps debug vs production identical at the config level: only environment variables change.

Key parameters:

- `PER_DEVICE_BATCH_SIZE` — primary handle on *memory usage per GPU* and *step-level noise*.
- `GRADIENT_ACCUMULATION_STEPS` — trades memory vs latency by amortizing optimizer updates over multiple forward/backward micro-steps.
- `SLURM_JOB_NUM_NODES` / `get_world_size()` — define the degree of data parallelism.

This design aligns with best practice in large batch distributed training, where global batch and LR are tied through simple scaling rules.

Training horizon

Purpose: Decide how long we train and where a resumed job should continue.

Technique: We use an **epoch-bounded** horizon with an optional token cap:

```
max_duration:
  tokens: -1       # disabled
  epochs: 3
hard_stop: -1      # no forced step limit
load_path: ${oc.env:LOAD_PATH, null}
```

How it works in our stack:

- When `tokens < 0`, we interpret `max_duration` as `Duration.epochs(epochs)` and let the data loader define epoch length.

- We expose `load_path` via the environment and set it from our SLURM wrapper (`train.sh`) based on the latest checkpoint, so resume is always an explicit decision, never implicit state.

- For early experimental phases, epoch-bounded runs are convenient because data ingestion and restart behavior are still evolving; later we can switch to token-bounded limits for cross-mixture comparability, which is standard in scaling-law analyses [1].

Key parameters:

- `epochs` — coarse control; used while pipeline and restarts are being hardened.
- `tokens` — fine-grained, disabled by default but compatible with scaling-law accounting.
- `hard_stop` — optional “circuit breaker” in steps (e.g., for safety or A/B testing).

Optimizer: AdamW with stability-oriented defaults

Purpose: Choose an optimizer that is robust on long-horizon, large-scale LM pre-training.

Technique: We use AdamW (decoupled weight decay) with gradient clipping:

```
learning_rate: 1e-4
weight_decay: 0.1
betas: [0.9, 0.95]
max_grad_norm: 1.0
z_loss_multiplier: 0
```

- **AdamW** decouples weight decay from the gradient step, addressing issues identified in adaptive optimizers using naïve L2 regularization [2].
- **Gradient norm clipping** is a standard remedy for exploding gradients [3].

How it works in our stack:

Configured via `SkipStepAdamWConfig`, which supports:

- `group_overrides` — we use this to disable weight decay on embeddings, a common practice to avoid shrinking embedding norms and destabilizing normalization layers.
- optional “skip-step” behavior if numerical problems are detected.
- `max_grad_norm=1.0` is enforced inside the train module; this is cheap protection against catastrophic single-step updates in long runs.
- `z_loss_multiplier` is wired but set to zero; we keep it available because small z-loss terms have been reported to stabilize Transformer training in very large LMs [4].

Key parameters:

- `learning_rate` — main convergence speed knob; tuned in conjunction with global batch.
- `weight_decay` — controls implicit regularization; decoupled from LR under AdamW.
- `betas` — momentum/variance smoothing; [0.9, 0.95] balances adaptivity vs noise.
- `max_grad_norm` — stability guardrail.
- `z_loss_multiplier` — disabled by default; reserved for future stability tuning.

LR schedule: Warmup–Stable–Decay (WSD)

Purpose: Shape how aggressively the optimizer explores the loss landscape across compute budget.

Technique: We use a Warmup–Stable–Decay (WSD) schedule:

```
scheduler:
  type: WSD
  warmup_min_lr: 3e-5
  decay_min_lr: 3e-5
  warmup_steps: 0
  warmup_delay: 0
  decay_steps: 0
```

WSD is designed for long-horizon pre-training: maintain a large, stable LR, then branch into a rapid decay phase to harvest a strong final checkpoint once a compute budget is chosen. Recent work explains its effectiveness via a “river valley” loss landscape model [5].

How it works in our stack:

During early infrastructure bring-up, we run with a stable LR (no decay) to focus on system correctness and throughput.

For the annealing phase (later in training), we enable decay_steps and decay_fraction to implement a trapezoid-like schedule: constant LR followed by a triangular decay to `decay_min_lr`.

This neatly aligns annealing with data mixture changes (e.g. specialized late-stage curricula) and wall time constraints.

Key parameters:

- `warmup_steps`, `warmup_min_lr` — if enabled, control the initial ramp-up, mitigating optimization issues common in large-batch regimes.
- `decay_steps`, `decay_fraction`, `decay_min_lr` — determine how quickly we “exit the river valley” and settle into lower LR.
- `warmup_delay` — allows postponing warmup, e.g. when resuming mid-run.

It lets us run at a stable LR while we validate infrastructure, then branch into annealing without needing to predetermine the full step budget from day one.

It makes it easier to align annealing windows with operational constraints (walltime, data phase boundaries), aligning our decision to use a Trapezoid learning rate schedule. For more information, please see Section 2.3.

Compute path accelerators: compilation + FlashAttention

Purpose: Reduce per-token latency and memory overhead so that a 26B-scale model fits and runs efficiently on multi-node H100.

Technique:

```
model_compile: true
use_flash_attention_2: true
```

`model_compile: true` enables TorchInductor (torch.compile) to fuse kernels and optimize the compute graph.

`use_flash_attention_2: true` activates FlashAttention-2, an IO-aware exact attention implementation that reduces expensive HBM<->SRAM traffic via tiling [6,7].

How it works in our stack:

We manage **per-job/per-rank caches** (`TORCHINDUCTOR_CACHE_DIR, TRITON_CACHE_DIR, XDG_CACHE_HOME`) and wipe them at job start, to avoid stale compiled artifacts.

We ensure debug runs follow the same compile and attention paths as production, so we catch kernel-specific bugs early.

Key parameters:

- `model_compile` — toggles the compiler; primarily affects first-step latency and long-run throughput.
- `use_flash_attention_2` — trades some implementation complexity for substantial speedups and memory savings at long sequence lengths.

2.4.2 Phase B - Launching jobs

Top-level flow: from `sbatch` to `trainer.fit()`

Purpose: This structure exists because we want one single training entrypoint (`scripts/snellius/train.sh $CONFIG_PATH`) and make all operational behavior (resume, profiling, walltime handling, restart policy) a wrapper concern.

Technique: At a high level we run:

```
sbatch launch_train_prod.job (or launch_train_debug.job)
  -> srun scripts/snellius/train.sh
      -> set up modules + venv + caches + MONITORING TOOLS
      -> resolve checkpoint dir + LOAD_PATH
      -> optional GPU health check / optional nsys profiling
      -> torchrun (multi-node rendezvous via c10d)
      -> on exit: handle_restart (state file + optional resubmit)
```

Production gating: code provenance guarantees

Purpose: Ensure all nodes run exactly the same code revision.

- no uncommitted changes `git status --porcelain` excluding untracked
- branch is main
- local commit equals `origin/main` (via git fetch + comparing SHAs)

Why we do it: Distributed training amplifies small inconsistencies if even one node uses slightly different code, we observed that we can get non-reproducible failures. This gating makes our run auditable and prevents accidentally *dirty tree* launches.

We also provide `deploy_to_shared.sh` which rsyncs only git-tracked files to a shared path. So, it enforces a consistent code snapshot across nodes and avoids shipping local ephemeral state.

SLURM resource shape: debug vs production

For our production we start long and multi-node runs while for debug we start short and single-node ones, using the SLURM's `sbatch` command [8].

Production (plunch_train_prod.job):

```
--partition=gpu_h100
--nodes=22
```

```
--gpus-per-node=4
--ntasks-per-node=1
--time=5-0 (5 days)
--exclusive (avoid noisy neighbours)
--signal=TERM@60 (send SIGTERM 60s before end)
```

Debug (pretrain_debug.sh):

```
same partition and 4 GPUs, but
--nodes=1 and
--time=0:20:00
```

Why we do it: debug should validate correctness (imports, compilation, rendezvous, checkpoint logic) quickly, while production maximizes steady-state throughput.

We rely on `--signal=TERM@60` to receive a SIGTERM 60 seconds before wall time, giving us an opportunity to checkpoint and exit gracefully. This aligns with SLURM's recommended pattern for cleanup logic.

Key parameters:

- `--nodes`, `--gpus-per-node`, `--ntasks-per-node` — define distributed topology.
- `--time` — caps wall time; coupled to auto-restart logic.
- `--exclusive` — minimizes noisy neighbours, important for consistent throughput.

Environment and dependency control

Purpose: Ensure a reproducible runtime environment across nodes.

Technique: Inside `train.sh` we:

- `module purge` then `module load 2024 NCCL/2.22.3-GCCcore-13.3.0-CUDA-12.6.0`
- activate our local venv
- set `OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK:-8}`

Why it matters: we want deterministic NCCL/CUDA pairings. Purging modules avoids inherited environment contamination, while we load specific versioned libraries.

Distributed rendezvous: torchrun + c10d, with explicit master host/port

Purpose: Launch the multi-process, multi-node job and form a global process group.

Technique: We choose:

- `MASTER_ADDR=$(scontrol show hostnames | head -n 1)`
- `MASTER_PORT=39591`

and run our training with the distributed execution command:

```
torchrun \
  --nproc_per_node=$SLURM_GPUS_PER_NODE \
  --nnodes=$SLURM_NNODES \
  --rdzv_id=$SLURM_JOBID \
  --rdzv_backend=c10d \
  --rdzv_endpoint=$MASTER_ADDR:$MASTER_PORT \
  --master_addr=$MASTER_ADDR \
  --master_port=$MASTER_PORT \
  scripts/train.py $CONFIG_PATH
```

Key parameters:

- `rdzv_id` — uniquely identifies a worker group.
- `rdzv_backend=c10d, rdzv_endpoint` — define how workers discover each other.
- `master_addr, master_port` — conventional process-group configuration for torch.distributed.

This is aligned with PyTorch Elastic's documented rendezvous model: `rdzv_backend=c10d` and `rdzv_endpoint=<host>:<port>` define where workers coordinate to form the process group [4].

Why we do it: it makes multi-node startup explicit and debuggable; when something fails, the endpoint and rendezvous ID are visible in logs and can be correlated across nodes.

Cache discipline: isolate and purge compile/profiler caches per rank

Purpose: Avoid stale or corrupted compilation artifacts across restarts and code changes.

Technique: We set per-job/per-rank cache paths:

- `TMPDIR=$CACHE_DIR/cache/$SLURM_JOBID`
- `ORCHINDUCTOR_CACHE_DIR=..._$SLURM_NODEID_$SLURM_PROCID`
- `TRITON_CACHE_DIR=..._$SLURM_PROCID`
- `XDG_CACHE_HOME=..._$SLURM_PROCID`

and purge them before launch.

Why we do it: torch.compile and Triton generate artifacts that can become corrupted or incompatible across code changes. Isolating caches reduces heisenbugs and avoids cross-job cache poisoning—especially important when running many restarts like in our case (at least every 5 days).

MONITORING TOOLS wiring: project naming by mode & mode separation

Purpose: Keep debug and production metrics logically separated while sharing infrastructure.

Technique:

- `WANDB_PROJECT="GPT-NL-$MODEL_SIZE-train"` for production
- `WANDB_PROJECT="GPT-NL-$MODEL_SIZE-train-$TRAIN_MODE"` for debug
- `WANDB_MODE=online`

Why we do it: it keeps debug runs from polluting production dashboards while still exercising the full telemetry path.

Checkpoint directory resolution and resume policy

Purpose: Make *resumption* robust and predictable across restarts and node counts.

Technique:

11. We compute a default checkpoint directory per run that maps to the `SLURM_JOB_ID` so its unique (see code below)
12. Consult the **state file** (see B8) to check whether there is an existing run to resume from; if so, override `CHECKPOINT_DIR`.
13. Discover the latest `step<N>` subdirectory and export `LOAD_PATH` accordingly, in case OLMo-core's latest symlink is missing (see code below)

```
# point 1
CHECKPOINT_DIR="$PROJECT_SPACE/$MODEL_NAME-nodes-$SLURM_NNODES-mbs-$PER_DEV
ICE_BATCH_SIZE-gas-$GRADIENT_ACCUMULATION_STEPS-$SLURM_JOB_ID"

# point 3
LATEST_CHECKPOINT=$(ls "$CHECKPOINT_DIR" | grep '^step[0-9]\+$' | ... | tai
l -n1)
export LOAD_PATH="$CHECKPOINT_DIR/step$LATEST_CHECKPOINT"
```

In our configuration we further set the steps that we store checkpoints:

```
save_interval: 880
ephemeral_save_interval: 110
save_async: true
```

Why we do it: we bias toward resume correctness (perfect resume without data repetition across several node counts) by checkpointing on carefully chosen step multiples, and we add ephemeral checkpoints as a higher-frequency safety net. OLMo-core's checkpoint callback is designed for exactly this pattern: permanent amd ephemeral intervals.

State file + auto-restart: training as a resumable workflow

Purpose: Decouple the logical training run from individual SLURM jobs.

Technique:

- `train_state_utils.sh` maintains a JSON state keyed on `(user, model, nodes, per_device_batch_size, grad_accum)` and tracks:
    - `checkpoint_dir`, existence
    - `job_id, runtime, status, timestamp`
    - `restart_count`
- `handle_restart(exit_code, checkpoint_dir, script_path, max_restarts)`:
    - Exit 0 -> mark `completed` and optionally stop.
    - Exit 1 -> mark `failed` and, if allowed, re-queue.
    - Other -> mark `failed` and require manual intervention.

Why we do it: This pattern mirrors standard HPC practices for managing long-running work-flows over multiple jobs. SLURM jobs are ephemeral; the training run is the persistent entity. The state file makes that persistence explicit and supports. At scale, transient failures (file system hiccups, node faults, scheduler preemption) are expected; automatic resubmission reduces operator toil and shrinks dead time.

Time-aware graceful termination

Purpose: Distinguish wall time preemption from manual cancellation and decide whether to restart.

Technique:

- In production, we install `trap graceful_exit TERM`.

- On SIGTERM:
    1. Query `TimeLimit` via `scontrol show job`.
    2. Compute remaining seconds.
    3. If close to zero, treat as walltime preemption (exit 0).
    4. Otherwise, treat as manual cancel (exit 1).
- Then delegate to `handle_restart`.

Why we do it: SLURM typically sends SIGTERM before SIGKILL; we use that window to exit cleanly and optionally restart, rather than losing progress to SIGKILL. This is consistent with SLURM's recommendation to use pre-kill signals for job cleanup.

GPU health check

Purpose: Fail fast if requested GPUs are degraded (thermal issues, ECC errors, or interconnect problems).

Technique:

- When `--gpu-health-check` is enabled, we:
    - Run an Apptainer (`.sif`) image across nodes, performing GPU stress and diagnostic tests.
    - Parse the container's summary and abort the run if any GPU fails.

Profiling: Nsight Systems traces on demand

Purpose: Obtain detailed CPU/GPU timeline traces for throughput bottleneck analysis.

Technique: If `--profiling` is set, we wrap the launcher with:

```
nsys profile --stats=true --trace=cuda \
  --cuda-memory-usage=true \
  -o traces/trace_${SLURM_JOBID} \
  $TORCHLAUNCHER
```

NVIDIA Nsight Systems is a system-wide performance analysis tool designed to identify bottlenecks across CPUs and GPUs.

Why we do it: During early experimentation or debug phases we wanted to check the usage of our GPU Nodes with more granularity. Still, Nsight Systems tracing can add non-trivial overhead, especially with broad CUDA API tracing and memory tracking enabled, so we use it selectively in short debug runs rather than in every production job [10].

2.4.3 Phase C -: Monitoring

Following OLMo-core's training module which is designed for async metric logging and flexible callbacks, we treat monitoring as *first-class control plane*, not an afterthought.

Metric collection strategy: planes and namespaces

Purpose: Organize metrics so we can rapidly distinguish between optimization, system, and workflow issues.

Technique: We group metrics into:

1. **Training dynamics (model health):**
    - loss curve stability, LR, grad norms, clipping activity
    - divergence indicators around schedule transitions (stable → decay)
2. **System efficiency (throughput health):**
    - tokens/sec and step-time variance
    - Dataloader stalls vs comm stalls vs compile regressions
3. **Run continuity (resumability health):**
    - time since last permanent and ephemeral checkpoint
    - correctness of resume (no data repetition, consistent step counters)
    - restart loop behavior (state file + restart count)

How it works in our stack: We record metrics in the train module (loss, lb, grad norm, etc.) and allow OLMo-core to gather/reduce them across ranks. Train modules are explicitly responsible for recording core metrics via `record_metric()` / `record_ce_loss()`, with optional namespaces and reduction behavior.

Why we do it: long-running pre-training fails in predictable ways; separating failure modes by plane lets us set sharper alerts and faster root-cause.

Throughput + memory monitoring

Purpose: Detect performance regressions and emerging OOM risks.

Technique: We rely on built-ins:

`SpeedMonitorCallback:` monitors throughput and is automatically added if not configured (we still usually configure it explicitly so dashboards stay consistent across runs).

`GPUMemoryMonitorCallback:` adds GPU memory statistics as metrics.

This combination lets us catch:

- Dataloader stalls (tokens/sec collapses, `batch_load_time` rises)
- silent OOM risk (allocated/reserved creeping up)
- interconnect regressions (step time rises while compute stays flat)

In-loop eval monitoring (lightweight guardrails)

Purpose: Detect capability drift and data/recipe regressions without expensive external evaluation.

Technique: We run periodic in-loop evaluations using the evaluator callback framework:

- `EvaluatorCallback` runs evaluators at a specified interval.
- `LMEvaluatorCallbackConfig` and `DownstreamEvaluatorCallbackConfig` configure common evals.

We treat eval as a drift detector, not a leaderboard generator:

- small, fixed validation sets for perplexity slope
- a few targeted tasks and benchmarks (e.g. arc_challenge_test_rc_5shot, hellaswag-nl_rc_0shot, etc.) to detect capability regressions after recipe changes

2.4.4 Phase D - Logging

Logging is where monitoring data becomes durable evidence. We log at three layers: SLURM, state files, and training metrics/artifacts.

SLURM logs are centralized and symbolically linked back into the repo workspace

Purpose: Preserve complete job stdout/stderr and make it easy to find per job.

Technique: Both debug and prod send logs to:

```
--output=/projects/prjs0986/wp14/olmo-logs/%j.out
--error=/projects/prjs0986/wp14/olmo-logs/%j.err
```

Then we create symlinks:

```
ln -sf /projects/olmo-logs/${SLURM_JOB_ID}.out $SLURM_SUBMIT_DIR/logs/${SLU
RM_JOB_ID}.out
ln -sf /projects/olmo-logs/${SLURM_JOB_ID}.err $SLURM_SUBMIT_DIR/logs/${SLU
RM_JOB_ID}.err
```

Why we do it: centralized storage avoids node-local loss; local symlinks make it easy for us to find the right logs from the project directory of our WP.

State logging

Purpose: Provide a single, structured source of truth for each logical run.

Technique: We write a single state JSON per logical run key and update it on transitions:

The state file encodes:

- `checkpoint_dir`, existence
- `job_id`, `hostname`, `num_nodes`
- `runtime`, `status` (`starting`, `running`, `completed`, `failed`, `max_restarts_reached`)
- `restart_count`

We update it on every transition and archive it when runs finish or are exhausted.

Why we do it: This way we create a user-friendly, machine-readable source of truth. It is intentionally append-free so that external automation can simply read the latest state without parsing logs.

Distributed logging hygiene (rank filtering + warnings everywhere)

Purpose: Avoid log storms from thousands of ranks while keeping critical messages visible.

We initialize the training environment with OLMo-core's `prepare_training_environment()`, which sets up distributed process groups and supports mixed backends (cpu:gloo,cuda:nccl). We intentionally keep a CPU backend available so that async checkpointing and bookkeeping collectives do not block training compute.

We also use `log_filter_type` semantics so that only selected ranks emit verbose logs (while warnings/errors always surface).

Console logging

Purpose: Provide quick, at-a-glance progress information without dashboards.

Technique: We use the console logger callback patterns so that:

- step-level progress is visible without opening dashboards
- periodic metric summaries are emitted at a controlled interval

This is coded in the callbacks API (see `ConsoleLoggerCallback` and the callback lifecycle hooks).

Experiment trackers and monitoring tools semantics

Purpose: Create a rich, query enabled history of experiments: metrics, configs, and artifacts [11].

Technique:

- We attach `WandBCallback` to the trainer with:
  - `enabled=True` in production.
  - `name=config_yaml["run_name"]`.
- Monitoring tools log metrics every step from rank 0 and attaches configuration dictionaries for full reproducibility.

Implication. Even if `metrics_collect_interval` is > 1, the tracker still sees dense curves; we therefore keep the metric vocabulary compact to avoid excessive volume.

Checkpoint logging & retention

Purpose: Balance compute loss (since last checkpoint) vs storage and IO overhead, while supporting flexible resumption.

Technique: We configure checkpointing as a logging artifact pipeline:

```
CheckpointerCallback(
    save_interval=save_interval,
    ephemeral_save_interval=ephemeral_save_interval,
    save_async=True,)
```

- Permanent checkpoints for every 880 steps provide stable rollback points and are intended to be retained long term.
- Ephemeral checkpoints for every 110 steps provide fine-grained restart points but can be pruned aggressively once upstream runs are healthy.

We heavily use async checkpointing and more specific ephemeral checkpoints that are intended for frequent recovery points (every 110 steps) and are perfect for resuming on (22, 11, 10, 5, 2) nodes.

2.4.5 References

[0] OLMO-core

[1] Scaling Laws for Neural Language Models

[2] Decoupled Weight Decay Regularization

[3] On the difficulty of training Recurrent Neural Networks

[4] 2 OLMo 2 Furious

[5] Understanding Warmup-Stable-Decay Learning Rates: A River Valley Loss Landscape Perspective

[6] FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning

[7] FLASHATTENTION: Fast and Memory-Efficient Exact Attention with IO-Awareness

[8] SLURM sbatch

[9] PyTorch torchrun

[10] NVIDIA Nsight Systems

[11] Experiment Tracking

2.5 Evaluation

During pre-training of the GPT-NL model, we evaluate its performance on both the training objective (next-word prediction) as well as downstream tasks, such as reasoning and reading comprehension.

During training, we continuously monitor model performance using two primary metrics:

- **Cross-entropy loss**: the average negative log-likelihood of predicted token probabilities, tracked on the training set (training dynamics) and validation set (generalization).
- **Perplexity**: the exponential of cross-entropy loss, representing how surprised the model is by actual next tokens in unseen text.

The graph below shows cross-entropy training loss across the entire trajectory, with phases color-coded for clarity. We observe consistent downward trends indicating stable dynamics, with occasional spikes that the model overcame autonomously. Key observations include: (1) a noticeable jump between Epoch 1 and Epoch 2 due to intentional data distribution changes, (2) steeper loss decreases during annealing phases from reduced learning rates, which extracts final performance gains, and (3) the end-of-Epoch-2 checkpoint was annealed mid-training for intermediate checkpoints, while Epoch 3 runs in parallel with Epoch 2 annealing to maintain progression.

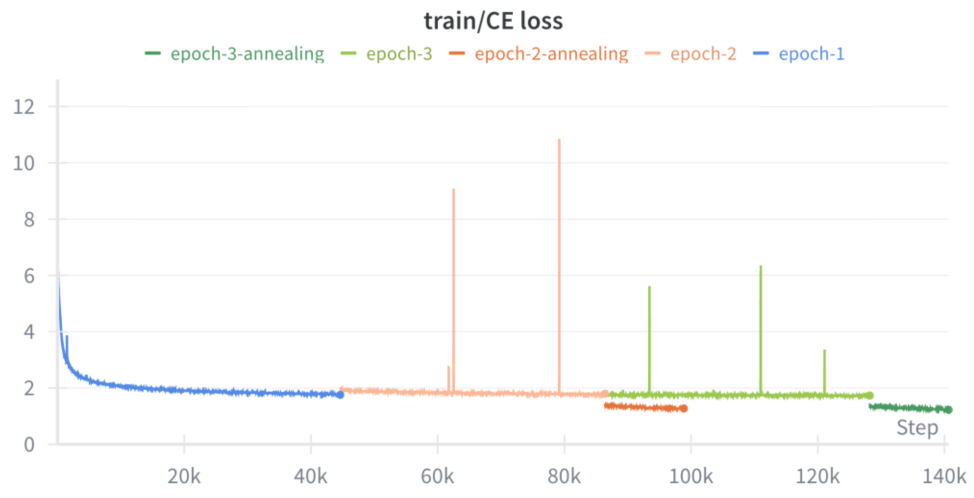
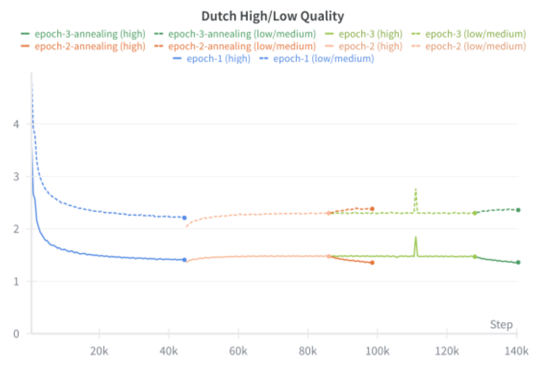


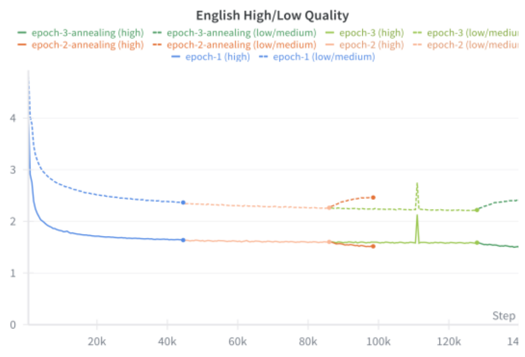
Figure 10: Pre-training loss

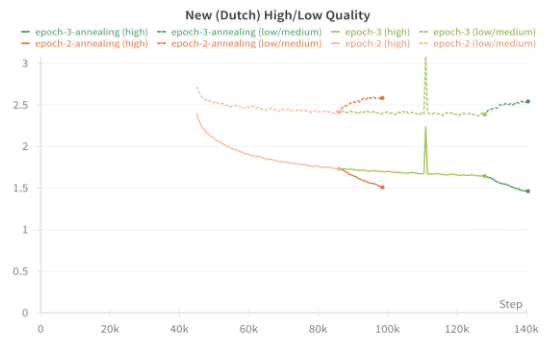
We further monitor validation cross-entropy separately for each language and quality bucket to track performance across our diverse data distribution, with Dutch and English subdivided by quality levels.

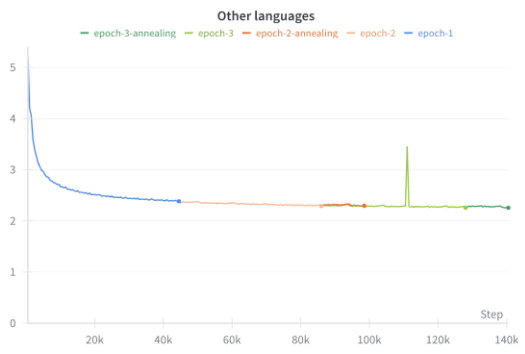
Epoch 1 to Epoch 2 transition: A clear inflection point occurs when new high-quality Dutch data and curated code data (with longer sequences) were incorporated mid-training. The Code bucket shows the most dramatic effect—loss drops sharply, indicating the model rapidly adapts to the longer, curated samples. The Dutch high-quality data similarly benefits from this shift. The New Dutch bucket, however, experiences a sudden improvement trajectory, accelerating its decline as the proportion of these sources increases in the training mix.

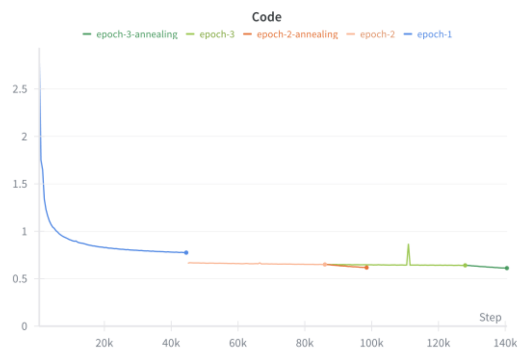
Annealing phases: When transitioning to annealing phases (both epoch-2-annealing and epoch-3-annealing), we deliberately shift toward high-quality sources only. This creates an interesting divergence: loss increases on low/medium-quality data (the model becomes less confident on lower-quality text), while it continues decreasing on high-quality data. This is the intended behavior—biasing the model toward generating high-quality output. The spikes visible in several buckets during annealing correspond to this intentional data distribution shift.











2.5.1 In-loop evaluation

To evaluate downstream task performance during pre-training, we implement the in-loop evaluation mechanism from OLMo, adapted to our training system (Groeneveld, Dirk, et al.). Rather than static post-training audits, this dynamic approach enables real-time issue detection. Our benchmarks assess **Reasoning and Commonsense** (HellaSwag, PIQA) and **Language Understanding** (MMLU, ARC):

Evaluation Task Name	Description	Example Evaluation (input: output)	Reference
`arc_challenge_test_rc_5shot`	ARC Challenge: hard multiple-choice science QA, 5-shot setting	Input: five example QA pairs + new question. Output: predicted answer choice.	Clark et al., 2018
`arc_easy_test_rc_5shot`	ARC Easy: easier multiple-choice science QA, 5-shot setting	Input: five examples + a new ARC-Easy question -> model predicts answer.	Clark et al., 2018
`piqa_val_rc_5shot`	PIQA: physical commonsense multiple-choice, validation split, 5-shot	Input: five goal/choice examples + new goal. Output: chooses correct solution.	Bisk et al., 2020
`hellaswag-nl_rc_0shot`	HellaSwag: commonsense next-sentence inference, zero-shot setting	Input: context. Output: selects most plausible continuation.	Zellers et al., 2019
`mmlu-nl_stem_mc_5shot`	MMLU: multilingual multiple-choice exam-style questions, 5-shot	Input: five example Q-A pairs + a new multiple-choice question. Output: predicted choice.	Hendrycks et al., 2021

As there are no official Dutch versions of some of the tasks we are interested, we have used the machine translated ones for the Dutch language, namely Hellaswag-nl and MMLU-nl.

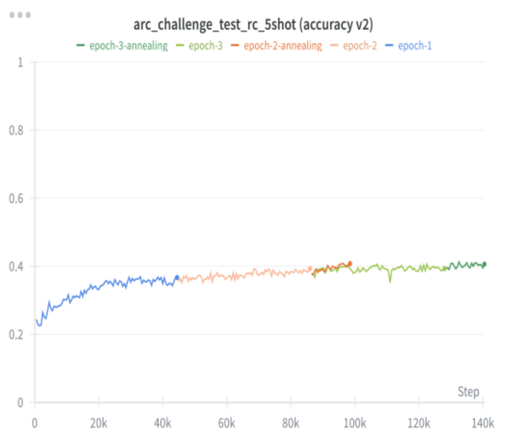
In addition, due to the big size of some of these tasks, we have developed a truncation mechanism and dynamically select part of the total available sets during our evaluations.

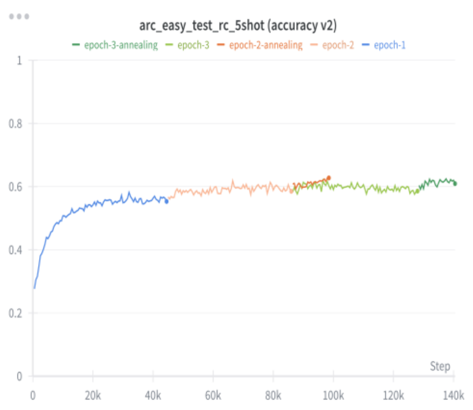
Setting	Definition	Example Prompt Structure
0-shot	The model is given only the task description (or question) with **no examples** beforehand.	*“Question: What is the capital of France? Choices: A) Berlin B) Paris C) Madrid D) Rome. Answer:”*
5-shot	The model is given **five examples** of the task with correct answers (few-shot learning) before the new test question.	*“Q1: … → A1: … \n Q2: … → A2: … \n … Q5: … → A5: … \n Now: Question: …”*

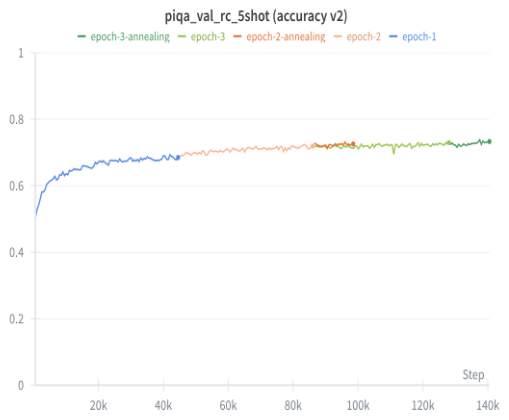
We employ both 0-shot and 5-shot settings to measure baseline generalization and assess in-context learning ability—whether the model can adapt to tasks without gradient updates.

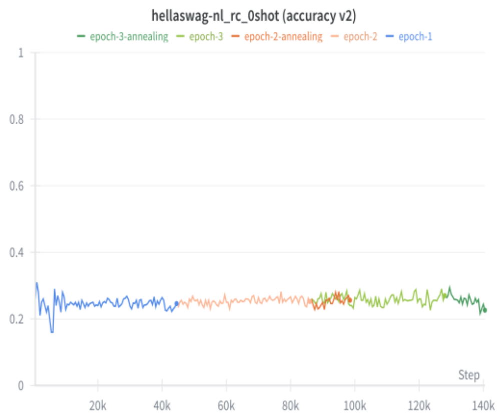
Knowledge and Commonsense Tasks: Results reveal distinct learning trajectories. ARC and PIQA show rapid initial improvement in Epoch 1 followed by convergence toward performance plateaus. The model achieves notably higher accuracy on ARC-Easy (~0.6) and PIQA (~0.8) versus ARC-Challenge (~0.4), indicating solid commonsense reasoning but limited scientific reasoning capability. These performance differences reflect task difficulty—easier tasks provide more reliable patterns in the pre-training distribution.

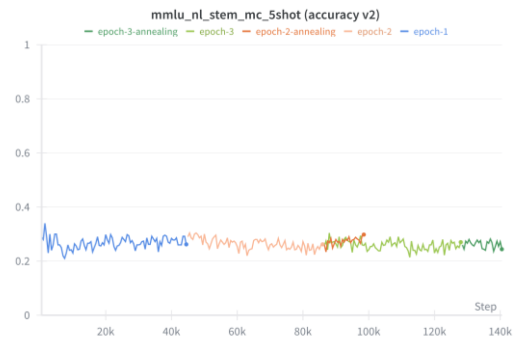
Reasoning Tasks: A critical limitation emerges for HellaSwag-NL and MMLU-NL, which remain near or below random baseline (0.25 for 4-choice) throughout training. HellaSwag-NL exhibits high variance early before stabilizing around chance levels, suggesting either insufficient reasoning capability or limited transfer from pre-training to complex multi-step reasoning. This knowledge-reasoning gap represents an important avenue for future improvements.











2.5.2 Out-of-loop evaluation

Out-of-loop evaluation is performed *after* pre-training has completed and does **not** influence the training process itself. Evaluation uses frozen model checkpoints and assesses downstream task performance, generalization, and alignment but do not affect training dynamics or model updates. We employ EuroEval as our offline benchmarking framework to comprehensively assess Dutch language performance. EuroEval provides a standardized, robust evaluation pipeline that:

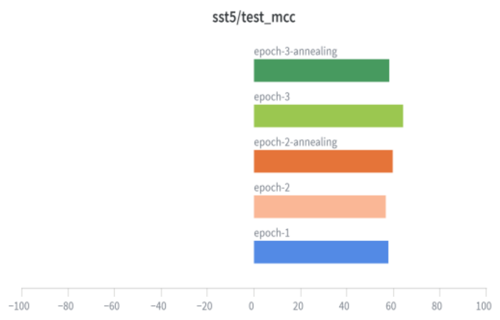
- Covers many task types relevant to Dutch, such as sentiment analysis, named-entity recognition, linguistic acceptability, reading comprehension, knowledge tasks, common-sense reasoning, and summarization.
- Uses bootstrapped evaluation, running each model–task pairing 10 times with resampled data and reporting mean scores with 95% confidence intervals, yielding statistically reliable performance estimates.

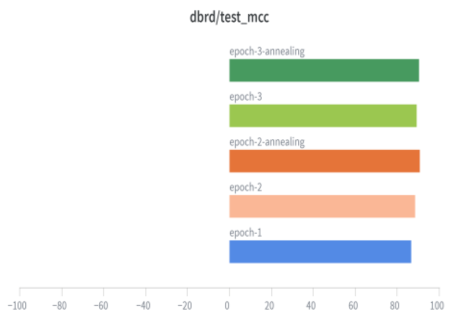
We run EuroEval periodically, selecting the latest checkpoint and comparing the progress of GPT-NL in various tasks.

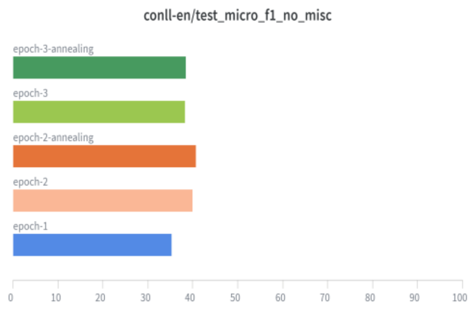
Below we list an overview of the Dutch tasks integrated into EuroEval:

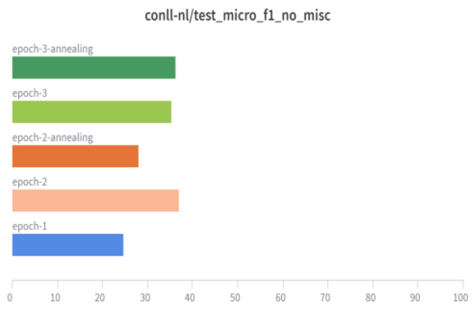
Task Category	Dutch Datasets	Evaluation Setup & Notes
Sentiment Classification	DBRD (Dutch book reviews)	Few-shot prompt (12 examples), generative sentiment label output (“positief/negatief/—”).
Named Entity Recognition	CoNLL-nl	Few-shot generative output as JSON dictionary of entities.
Linguistic Acceptability	ScaLA-nl, (Unofficial) Dutch CoLA	Few-shot prompts with “correct”/“incorrect” labels.
Reading Comprehension	SQuAD-nl, (Unofficial) BeleBele-nl, MultiWikiQA-nl	Generative answer output via prompt templates.
Knowledge	MMLU-nl, (Unofficial) ARC-nl	Few-/zero-shot question answering tasks.
Common-sense Reasoning	HellaSwag-nl, (Unofficial) GoldenSwag-nl	Select most plausible continuation via generative setup.
Summarization	WikiLingua-nl	Summarization of Dutch text, generative output.

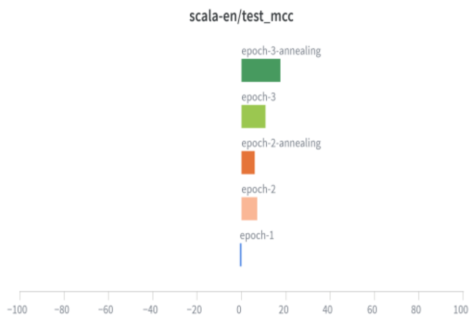
Within WP21, new Dutch benchmarks are in development that will be included in EuroEval, covering additional tasks like simplification and areas like bias.

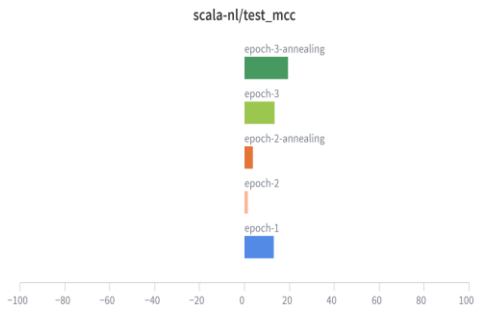


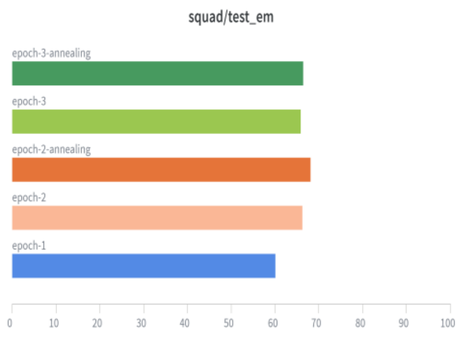


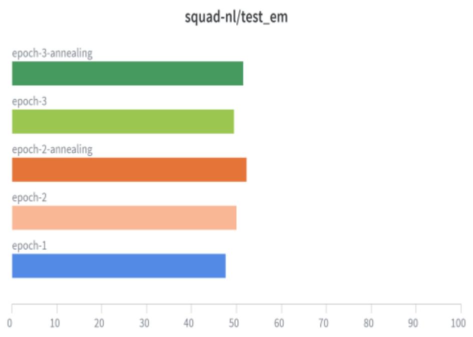


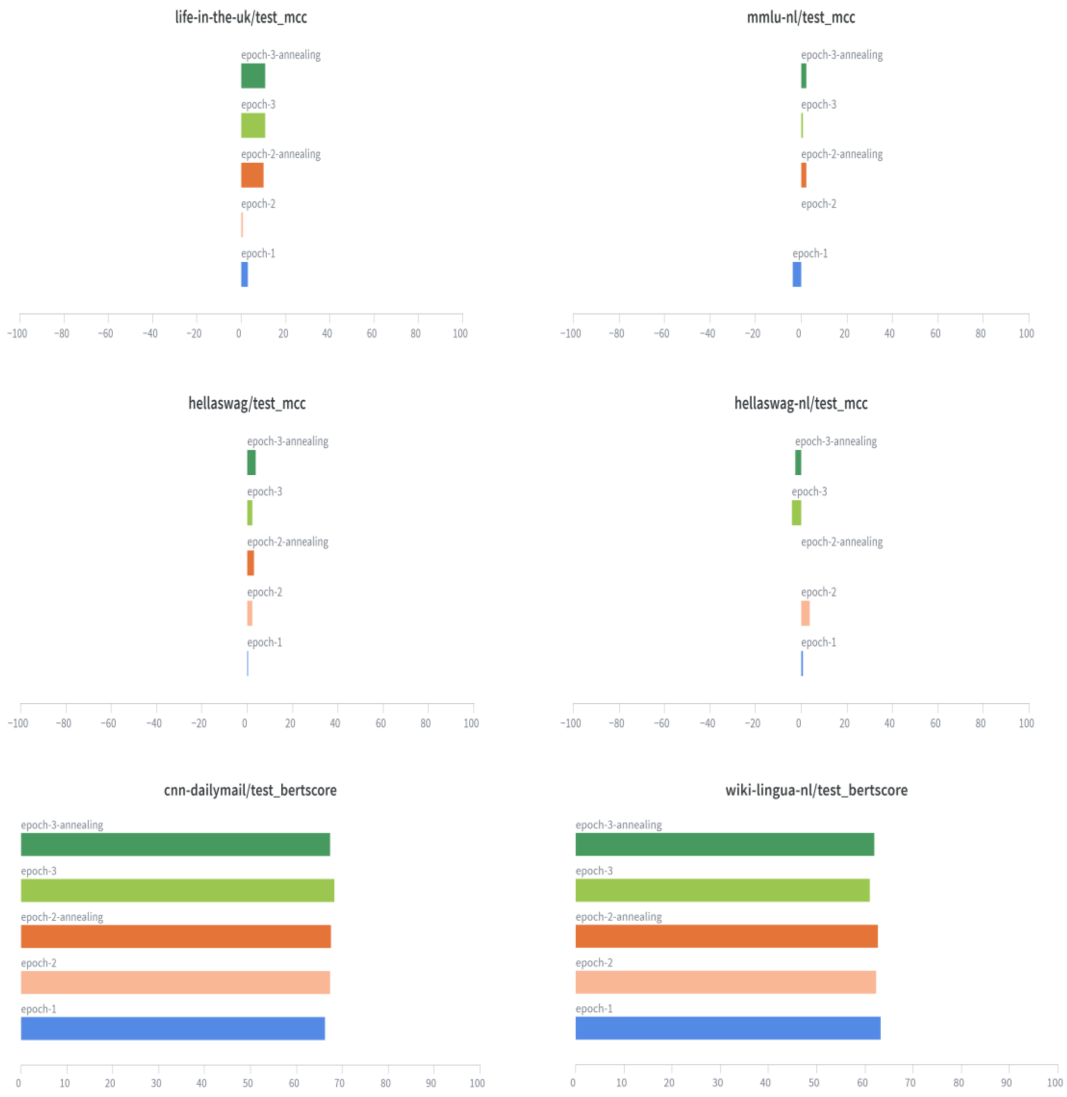












We can see from this table that generally later epochs have better results.

Sentiment classification

SST5 (EN) and DBRD (NL). We reach 90% on the Dutch DBRD, while the English stays at 58%.

Named Entity Recognition

Conll-en and Conll-nl have moderate results: 38% EN and 36% NL. There is a slight improvement from epoch 1 to the following epochs, but not a drastic one.

Linguistic acceptability

SCALA-EN and SCALA-NL. The metrics improve with the epochs but are very low (17% EN and 19% NL). This is one of the problematic tasks, as the model only has to output a letter as answer (multiple choice).

Reading comprehension

SQUAD and SQUAD-NL. Moderate performances: 66% EN and 51% NL.

Knowledge

LIFE-IN-THE-UK and MMLU-NL: knowledge. The metrics improve but are very low (10% EN and 2% NL). Also here, multiple choice is requested to the model.

Common sense reasoning

Hellaswag and Hellaswag-NL. For this task we have the worst results: 3% EN and -2% NL. For the Dutch version, the results for epoch-3 are also worse than the results for epoch-2.

Summarization

cnn-dailymail and wiki-lingua-nl. For this task, the results are acceptable: 67% EN and 61% NL.

2.6 Context Extension

For LLMs, the *context length* refers to the maximum number of tokens an LLM can process within a single forward pass. It determines how far back the model can look when interpreting or generating text. A longer context length enables the model to capture broader dependencies and maintain coherence across extended sequences [1].

GPT-NL is pretrained with a native context length of 4096 tokens. At a later stage, the development team included an effort to extend the supported context length using techniques such as RoPE Scaling and gradually increasing the context length in the pretraining and instruction fine-tuning phases.

Rotary Positional Embeddings (RoPE) encode relative positional information by rotating token representations in attention space [2]. RoPE scaling methods, such as NTK or YaRN [5] scaling, adjust the frequency of these rotations to allow extrapolation beyond the context lengths seen during training. While RoPE scaling improves numerical stability at longer contexts, it does not replace the need for exposure to long sequences during training.

2.6.1 Increasing Context Length

Extending a model's context length typically requires a continued pretraining at progressively larger sequence lengths, so the model learns long-range dependencies. Curriculum learning is commonly used, where the sequence length is increased in stages (for example, 4k to 8k to 16k) to improve stability and performance. Context lengthening usually occurs **during mid-training**, after the model has already learned short-range dependencies, ensuring that attention patterns can adapt to longer sequences without destabilizing previously learned knowledge. Depending on the positional encoding method, additional steps may include expanding positional embedding matrices or applying RoPE scaling after intermediate training phases [6].

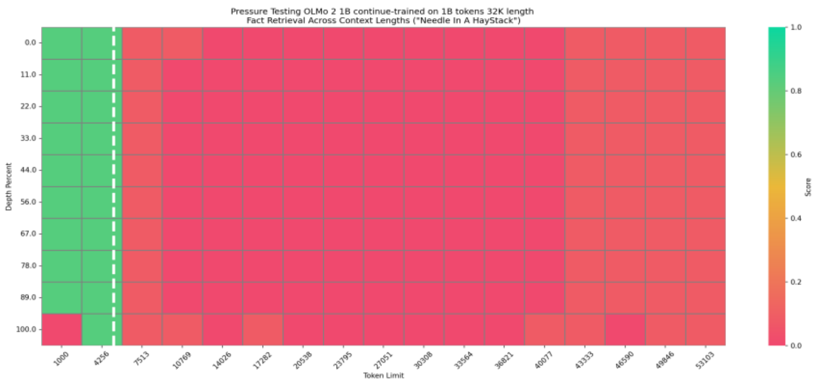
Within GPT-NL, the pretraining data mixture is split into a new subset of samples which are of 16k, 32k or 64k tokens. These subsets can be used for the curriculum learning-based mid-training.

Evaluation and Benchmarks

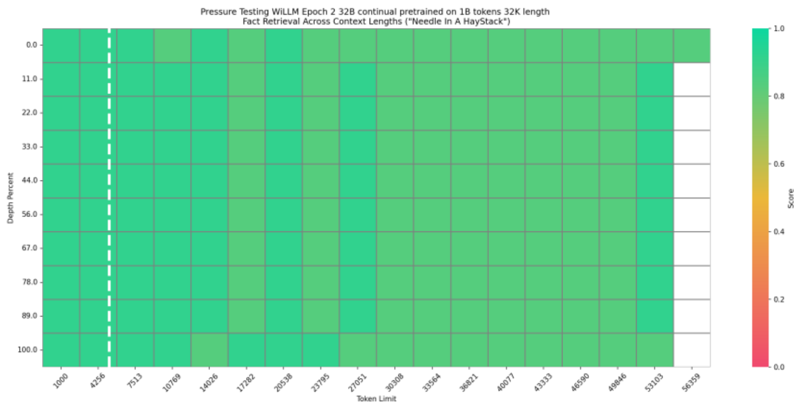
To evaluate GPT-NL capabilities and performance after Context Extension, we used 2 benchmarks:

- **Needle in the Haystack [4]:** A synthetic long-context retrieval benchmark that tests a model's ability to locate a specific piece of information (the "needle") within a large body of irrelevant text (the "haystack"), measuring basic long-range memory and recall performance.
- **RULER [3]:** A more comprehensive long-context evaluation suite that extends beyond simple retrieval to include multi-needle, multi-hop tracing, aggregation, and question answering tasks, aiming to assess a model's true long-context understanding capabilities as context length increases.

GPT-NL base:



GPT-NL with context extension:



Here we provide the complete results for both tasks: Needle in the Haystack (niah) and RULER.

Llama 3.1

Task	4096	8192	16384
niah_multikey_1	1.0000	1.0000	1.0000
niah_multikey_2	1.0000	1.0000	1.0000
niah_multikey_3	0.9900	0.9980	0.9900
niah_multiquery	0.9995	1.0000	1.0000
niah_multivalue	0.9910	0.9935	0.9930
niah_single_1	1.0000	1.0000	1.0000
niah_single_2	1.0000	1.0000	1.0000

niah_single_3	0.9960	0.9980	1.0000
ruler_cwe	0.9978	0.9776	0.6930
ruler_fwe	0.9620	0.9173	0.9660
ruler_qa_hotpot	0.6540	0.6300	0.5820
ruler_qa_squad	0.7710	0.7303	0.7090
ruler_vt	0.9996	1.0000	0.9992
Average	**0.9508**	**0.9419**	**0.9170**

GPT-NL

Task	4096	8192	16384
niah_multikey_1	0.9980	0.9960	0.9280
niah_multikey_2	0.9860	0.9900	0.9720
niah_multikey_3	0.6940	0.5000	0.5120
niah_multiquery	0.9790	0.9430	0.7860
niah_multivalue	0.6620	0.5690	0.4815
niah_single_1	1.0000	1.0000	0.9980
niah_single_2	1.0000	1.0000	1.0000
niah_single_3	0.7060	0.7600	0.6120
ruler_cwe	0.6438	0.3544	0.2354
ruler_fwe	0.7960	0.8080	0.8353
ruler_qa_hotpot	0.4100	0.3800	0.3480
ruler_qa_squad	0.5192	0.4138	0.4168
ruler_vt	0.9956	0.9896	0.9732
Average	**0.7992**	**0.7464**	**0.6999**

Performance comparison

Model / Metric	4096	8192	16384
GPT-NL	0.7992	0.7464 (6.61%)	0.6999 (12.45%)
Llama 3.1	0.9508	0.9419 (0.94%)	0.9170 (3.52%)

Remark: These results are using the epoch 2 weights, not the final model weights.

What all these tables mean:

- GPT-NL loses ~6.6% by 8K, and a total of ~12.5% by 16K (relative to 4K baseline).
- Llama 3.1 only loses ~0.9% at 8K, and ~3.5% at 16K.

Thus Llama 3.1 is significantly more robust to context scaling — its degradation is only a fraction of GPT-NL. In practical terms, Llama 3.1 retains high performance even when context is quadrupled in length, while GPT-NL degrades more noticeably.

To sum up:

- Adding dynamic RoPE scaling does not hurt performance and enables longer context performance

- The context lengthening performance is very inferior to any SOTA models, even smaller models from a couple of years ago
- Even on a native 4k context length our models perform bad (according to benchmark results). Finetuning does not have a significant negative effect on the context lengthening

2.6.2 References

[1] A Controlled Study on Long Context Extension and Generalization in LLMs

[2] LongRoPE: Extending LLM Context Window Beyond 2 Million Tokens

[3] RULER: What's the Real Context Size of Your Long-Context Language Models?

[4] Needle in the Haystack for Memory Based Large Language Models

[5] YaRN: Efficient Context Window Extension of Large Language Models

[6] Roformer: Enhanced transformer with rotary position embedding

2.7 Data folder Structure and Source Code Organization

The training set-up is based on the existing implementation of AI2's OLMo(-core). Initially we used the original OLMo codebase, but during development switched to the newest, optimized OLMO-core.

For our original comparison between using the HuggingFace Transformers framework and the AI2 OLMo framework, see the results of the November 2024 experiments. The OLMo training code, written in native PyTorch, showed a substantially better performance in terms of speed.

During our initial development, the OLMo-core package was published by AI2. Some initial tests comparing our setup using this framework versus using OLMo showed a ~20% speedup in throughput (measured in tokens/device/sec). This improvement in performance as well as the more up-to-date support and maintenance on the OLMo-core package led to the decision to switch framework.

The code base has been split up into several modules:

2.7.1 OLMo-Core

A fork from the original OLMo-core code base. This allows us to make small changes specific to our setup in a structured way. Changes include adding the GPT-NL tokenizer to the configuration, updating evaluation and conversion scripts.

```
/olmo-core
└── /docs
└── /src
    ├── /examples # training recipe examples
    └── /olmo_core
        ├── /data # scripts fro data handling
        ├── /distributed # scripts for distributed training
        ├── /eval # scripts for evaluation
        │
        ├── /float8 # scripts for precision handling
        ├── /internal # scripts for leaderboard ranking
        ├── /kernels # scripts for MoE kernel
        ├── /launch # scripts for training launchers
        ├── /nn # scripts for neural networks
        ├── /ops # scripts for MoE operations
        ├── /optim # scripts for optimization routines
        ├── /train # scripts for training routines
    ├── /scripts # various sizes training recipes
    ├── /test # test suite for olmo_core components
```

2.7.2 Pytorch native

All GPT-NL code, including scripts for running the distributed training, evaluation, inference along with installation and debugging scripts.

```
/pytorch_native
└── /config # model and train configuration
│   ├── gpt-nl-1B.yaml
│   ├── gpt-nl-26B.yaml
└── /logs # output folder of the jobs
│   ├── job_number.out
│   ├── job_number.err
└── /scripts
    ├── /conversion # checkpoint conversion recipe
    ├── /evaluation # checkpoint evaluation scripts
    ├── /inference # inference workflow scripts
    ├── /installation # installation scripts
    │   ├── install_olmo_snellius.sh
    │   ├── update_olmo_local_environment.sh
    └── /snellius # Training job scripts
    │   ├── .env
    │   ├── copy_to_archive.job
    │   ├── copy_to_snellius.job
    │   ├── deploy_to_project_space.sh
    │   ├── launch_train_debug.job
    │   ├── launch_train_prod.job
    │   ├── train.sh
    │   ├── train_state_utils.sh
    └── cli_helpers.sh
    └── train.py # main training logic
└── README.md
└── pyproject.toml
```

For more information on the Training workflow please look at Section 2.4.

2.7.3 OLMo-in-loop-evals

A fork from the original OLMo-in-loop-evals, updated to include Dutch benchmarks.

```
/olmo-in-loop-evals
└── /olmo_eval
```

```
        ├── /hf_datasets # local evaluation datasets
        └── /oe_eval_tasks # configuration files for eval tasks
        └── /tokenizers
        ├── metrics.py
        ├── tasks.py
        ├── tokenizer.py
        ├── util.py
        ├── version.py
    ├── /scripts # release scripts
    ├── /tests # test suite for eval tasks
    └── README.md
    └── pyproject.toml
```

2.7.4 Model conversion

During training, the model checkpoints are saved as a PyTorch model, as well as in a distributed format (.distcp). Since many downstream applications use the HuggingFace/Transformers library, we convert the model checkpoints to the HuggingFace-compatible safetensors format in bfloat16 precision.

2.7.5 Practical notes

2.7.5.1 Installation

It is adviced to clone all the repositories and start the installation following the below order:

This workflow assumes access to Snellius

```
cd pytorch_native
chmod +x scripts/installation/install_olmo_snellius.sh
bash scripts/installation/install_olmo_snellius.sh

source venv/bin/activate

cd ../olmo-core
pip install -e .

cd ../olmo-in-loop-evals
pip install -e .
```

2.7.5.2 Logs and workspace

All of our logs and checkpoints are store to a shared place `$PROJECT_SPACE` under the `OLMo-core` folder. There, each training job creates a folder with the format: `$PROJECT_SPACE/$MODEL_NAME-nodes-$SLURM_NNODES-mbs-$PER_DEVICE_BATCH_SIZE-gas-$GRADIENT_ACCUMULATION_STEPS-$SLURM_JOB_ID`, e.g. `gpt-nl-26B-nodes-22-mbs-12-gas-3-12967709`.

The folder has the below structure:

```
/OLMo-core
└── /checkpoint_step_number
└── ...
│       ├── /model_and_optim
│       ├── /train
│    ├── .metadata.json
│    ├── config.json
│    ├── data_paths.txt
└── /wandb
│    ├── /latest_run
│    ├── debug-internal.log
│    ├── debug.log
```

3 Architecture of the Instruction Fine-Tuning

In this set of pages, we describe the GPT-NL fine-tuning approach that we carried out between July and December 2025. We start with a brief motivation about why pre-training itself is not enough, which type of fine-tuning we employ and why. Then we give an overview of the fine-tuning process as we implement it and provide an outline of content in this chapter.

A pre-trained model is a raw language generator that is not useful yet: it does not follow instructions. It is optimised to find a likely next token (a numerical representation of words, or parts of words), given the previous tokens. What is likely to follow in a text, is not necessarily the most useful. Consider the example below, where the base model is asked in which year a particular film was released. The model produces a series of years, instead of formulating a coherent answer that provides a (single) answer to this question.

Example:

```
Q: In which year was The Godfather first released?
A: 1972 1974 1976 1978 1980 1982 1984 1986 1988 1990 1992
```

Modern post-training (i.e., everything that comes after pre-training) is a series of fine-tunes with different aims that build upon each other and require careful design:

- **Instruction fine-tuning** teach formatting and base of instruction following behaviour (e.g., chat interactions, answering questions)
- **Preference tuning**: align to human preferences (safety, tone of voice)
- **Reinforcement learning**: boost performance on verifiable tasks (e.g., math, precise formatting, reasoning)

As the resources for this activity in this stage of the GPT-NL project are constrained (datasets, compute, and time), we resort to instruction fine-tuning only for now.

3.1 GPT-NL Instruction fine-tuning

Let's start with a definition of instruction fine-tuning:

> *Instruction fine-tuning is defined as (most-often) supervised fine-tuning (SFT) on instruction-demonstration data, potentially in a conversational format. This type of training makes that the model can follow instructions and make (useful) predictions (potentially with CoT) in a zero-shot (or few-shot) setting.*

Supervised fine-tuning trains the model on instruction-response pairs (either single-turn prompt-completion or multi-turn conversational exchanges) by masking the input tokens and computing cross-entropy loss only on the output tokens (assistant responses), teaching the model to generate appropriate responses rather than predict any next token. This approach transforms the pre-trained language model from a raw text generator into an instruction-following assistant that produces coherent, task-oriented outputs.

Post-training can elicit various capabilities and behavioral traits in the model. Given the broad range of possible objectives, we have narrowed these down to a set of priority goals that are important, achievable through supervised fine-tuning, and for which data is available. We will refer to this set of objectives as the GPT-NL Priorities in figures. The table below recaps the objectives that made it to the final selection.

Type	Objective	Relative priority	Dataset available	Main strategy
Instruction Following	General instruction following	High	✓ Yes	✓ Yes
Instruction Following	Supporting chat-style interaction	Low	✓ Yes (OASST)	✓ Yes
Instruction Following	Precise formatting following (JSON)	Low	A bit SciRIFF	Maybe
NLP Tasks	Specializing in main GPT-NL NLP tasks	High	✓ Yes (GPT-NL IT dataset)	✓ Yes
NLP Tasks	RAG	High	Only contextual QA	✓ Yes
NLP Tasks	Generalizing to a broader set of GPT-NL tasks	Low	✓ Yes (FLAN)	✓ Yes
Long Context	Longer context processing than the base model (>4096)	Medium	Yes, if continual pre-training	✓ Yes
Knowledge	Establishing solid knowledge recall	Medium	✓ Substantial QA data	✓ Yes

Examples of objectives that did not make it to this priority list include multi-lingual capabilities like translation, precise instruction following (e.g., writing exactly three paragraphs when instructed), and safety objectives. The latter category encompasses multiple aspects, such as producing misinformation or disinformation as well as generating harmful content. While these aspects are important for the project, they are not included in this initial priority list because we do not have data available for these objectives and other types of fine-tuning (e.g., preference fine-tuning) might be more suitable for addressing them.

3.1.1 GPT-NL instruct dataset

GPT-NL set out to create its own Dutch instruction fine-tuning dataset, consisting of ~15K prompt-completion pairs. The dataset has been created by human annotators of one independent company, following detailed instructions. The dataset is subdivided in a few tasks:

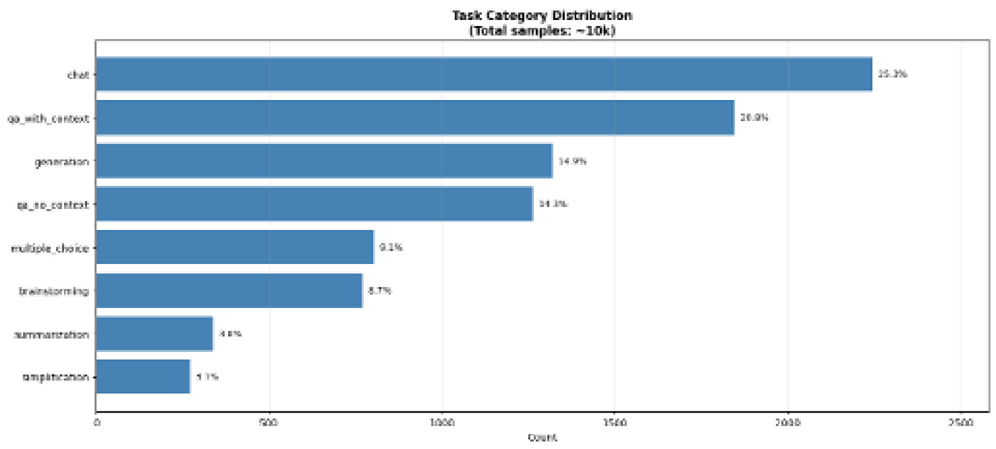


Figure 11: Task distribution GPT-NL Instruct Dataset

3.1.2 Overall instruction fine-tuning process

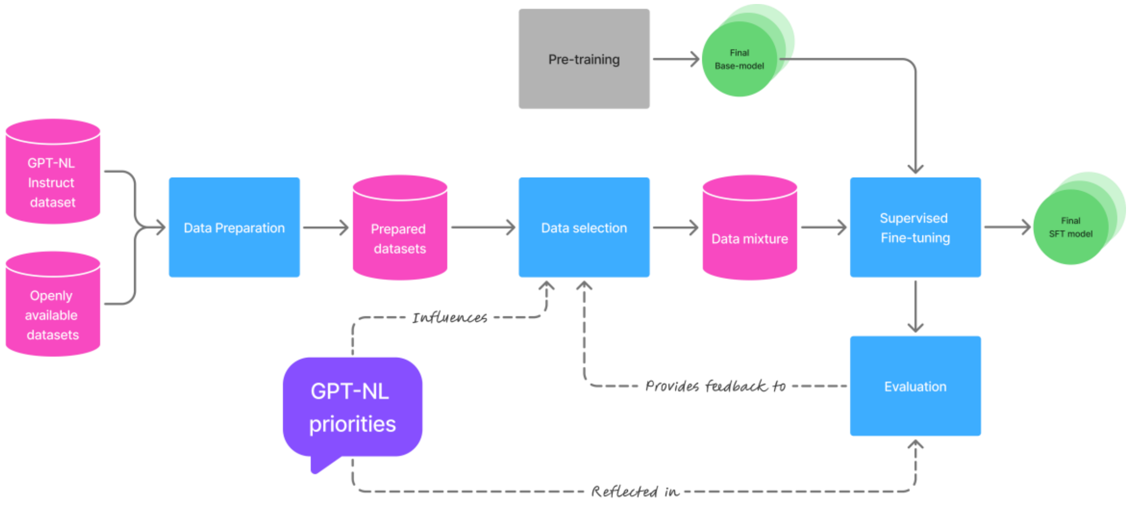


Figure 12: Fine-tuning Overview

The overall process (visualised in the diagram above) takes as input a pre-trained base model along with the GPT-NL instruction dataset and other openly available datasets, and produces an instruction fine-tuned model checkpoint. Since the pre-training pipeline may produce multiple versions of the base model, fine-tuning can be performed on different base checkpoints. Additionally, the fine-tuning pipeline experiments with different data selections for each base model, resulting in multiple fine-tuned variants per base checkpoint.

1. Data preparation - Preparing the datasets into proper and unified prompt-completion format
2. Data selection - Combining datasets in different proportions and filtering out parts of datasets
3. Training - Supervised Fine-Tuning (SFT) implementation that modifies the model weights, with configuration, execution on Snellius HPC, and monitoring capabilities

4. Evaluation - Evaluating model performance through internal and external benchmarks, comparing fine-tuned variants, and analysing results across task categories and languages

For a technical overview of how the codebase and data are organized, see Code and Data Organization.

3.2 Fine-Tuning Data Preparation

This section describes the data preparation process for GPT-NL instruction fine-tuning, which transforms diverse datasets into a unified format suitable for training. The pipeline handles dataset acquisition, standardization, filtering, and formatting to ensure consistent training data formats across multiple sources and languages.

3.2.1 Why data preparation is critical?

Raw datasets from different sources have inconsistent formats, varied quality, and different structures that make them unsuitable for direct use in fine-tuning. The data preparation pipeline addresses several key challenges:

3.2.1.1 Format Heterogeneity

Different datasets use incompatible schemas and field names. One dataset might use `"question"` and `"answer"` fields, while another uses `"input"` and `"output"`. Training frameworks like Hugging Face TRL require consistent formats with specific field names. Without standardization, training scripts would need custom handling for each source, leading to parsing errors and inability to batch samples efficiently. The pipeline transforms all datasets into a unified schema (`instruction`, `context`, `response`, `task_category`, etc.).

3.2.1.2 Quality and Noise in Crowdsourced Data

Datasets from online forums and crowdsourcing platforms (e.g., Goeievraag.nl, OpenAssistant) contain data of varying levels of quality including factually incorrect information, incomplete responses, off-topic discussions, platform-specific artifacts, and toxic language. Training on low-quality data causes models to reproduce incorrect information, develop poor instruction-following capabilities, and amplify harmful patterns. The pipeline applies multi-stage filtering using PII detection, toxicity screening, and LLM-as-a-judge evaluation to ensure only high-quality examples are used.

3.2.1.3 Task Distribution Imbalance

Raw dataset collections often have severe imbalances in task types. Without explicit tracking and balancing, models can over-optimize frequent tasks at the expense of rare but important capabilities. Explicit `task_category` labelling (including automated inference for datasets lacking categories) enables visibility into task distribution, strategic data selection, and targeted augmentation of underrepresented capabilities.

3.2.1.4 License Compliance and Data Provenance

Datasets may have restrictive licenses (e.g., CC BY-SA requiring share-alike), contain AI-generated content, or include content with unclear licensing terms. Using incompatible licensed data creates legal liability and may violate organizational policies. The pipeline implements systematic license review, filtering to approved licenses only (MIT, Apache-2.0, CC BY 4.0), and exclusion of LLM-generated content.

3.2.1.5 Conversation Structure Inconsistencies

Different datasets represent conversations in fundamentally different ways: single-turn Q&A, multi-turn dialogues with branching, instruction-response pairs without explicit roles, and complex nested contexts. Inconsistent handling leads to inefficient tokenization, incorrect training signals, and poor conversational performance. Specialized “unrolling” scripts transform diverse conversation structures into standardized role-based format (`user`/`assistant` messages).

3.2.1.6 Model-Specific Template Requirements

Modern instruction-tuned models expect specific formatting with special tokens (e.g., `<bos>`, `<start_of_turn>`, `<end_of_turn>`) that vary between model families. These templates define the training signals that teach models to distinguish between user inputs and model responses. Incorrect template applications cause models that cannot distinguish between turns, degraded instruction-following, and training instabilities. The pipeline uses tokenizer-based template application to ensure model-specific formatting is correctly applied.

3.2.2 Datasets

This is a table of the datasets used for instruction fine-tuning. The number of samples is representative of the amount of the dataset that we consider for use, after the processing pipeline, but before the data selection process. The actual size might be larger, depending on the dataset.

Dataset	Domain	Description	# Samples	Language	Task Categories
SciRIFF	Science	Instruction-following tasks for scientific literature understanding.	**99,194**	English	information_extraction, multiple_choice, qa_with_context, reasoning, summarization
Aqua RAT	Math	Math word problems with multiple choice answers and rationales.	**97,721**	English	reasoning
Open-Assistant (OASST 1)	General (practical, scientific, creative, etc.)	Crowd-sourced multi-turn conversations.	**25,224**	English	chat
Narrative QA	Fiction, Entertainment	QA pairs from books and movie scripts.	**18,083**	English	qa_with_context
Goeievraag.nl	General	Dutch Q&A forum similar to Quora.	**17,799**	Dutch	qa_no_context
Aya Dataset	General / Cultural	Multilingual instruction dataset.	**5,032**	English, Dutch	brainstorming, chat, generation,

					information_extraction, multiple_choice, qa_no_context, qa_with_context, reasoning, simplification, summarization
SciTLDR	Science	Extreme summaries of scientific papers.	**431**	English	summarization
Qasper Dataset	Science	QA dataset on scientific papers.	**1,175**	English	qa_with_context
FLAN (combined)	General	Reformatted NLU datasets for zero-shot & CoT prompting.	**129,176**	English	multiple_choice, reasoning, generation, qa_with_context
Huggingface H4	General	Small, handcrafted instruction dataset.	**248**	English	brainstorming, chat, generation, information_extraction, multiple_choice, qa_no_context, qa_with_context, reasoning, simplification, summarization
SPIN	General	Manually generated prompt/completion pairs.	**15,000**	Dutch	brainstorming, chat, generation, multiple_choice, qa_no_context, qa_with_context, simplification, summarization
— TOTAL —	—	—	**409,083**	—	—

3.2.2.1 Data Acquisition

The data has highlighted above has been acquired from various sources such as the online datasets on HuggingFace or acquired through license purchase as is the case for Goeievraag.nl. Based on the requirements set by the GPT-NL project, only open-sourced datasets with the following license: MIT, CC BY-SA 4.0 or APache-2.0 were taken into consideration. Furthermore, other criteria was that large language models must not generate the open-source datasets. Thus, taking this consideration into account, we were able to collect various datasets as listed above.

Regarding the Flan dataset (see table below), which is composed of multiple sub-datasets added iteratively over time, we selectively included only those that met the previously mentioned criteria. These include GSM8K, AQuA-RAT, StrategyQA, QASC, and CREAK. For the AQuA-RAT dataset specifically, we opted for the most recent version available, rather than the one listed in the reference table. As for the sub-datasets WinoGrande, Taskmaster, and Dialog, we were unable to identify coherent open-source versions that met the licensing and origin requirements. Consequently, these were excluded from the final instruction dataset. The table below shows a list of sub-datasets that were analysed for the FLAN Dataset. It must be noted that the Flan dataset contains additional sub-datasets that were not reviewed, as the current selection sufficiently meets the needs and requirements of this project.

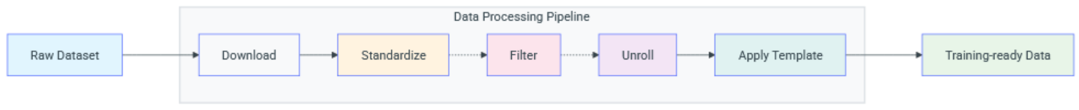
Subset	dataset	License	GPT-NL compatible	Notes
flan2021	SQuAD (v1/v2)	CC BY-SA 4.0	No	ShareAlike copyleft applies to derivatives/redis-tributions
flan2021	GSM8K	MIT	Yes	Permissive
flan2021	AQuA-RAT	Apache-2.0	Yes	Permissive
flan2021	QASC	CC BY 4.0	Yes	Attribution required not copyleft
flan2021	StrategyQA	MIT	Yes	Permissive
flan2021	e-SNLI	MIT (repo); SNLI base CC BY-SA 4.0	No	Underlying SNLI share-alike applies
flan2021	CREAK	MIT	Yes	Permissive
flan2021	ComVE (Sense-Making)	CC BY-SA 4.0	No	ShareAlike copyleft
flan2021	QED	CC BY-SA 3.0 / GFDL-derived	No	Wikipedia-derived; share-alike applies
t0 (P3/T0)	SNLI	CC BY-SA 4.0	No	ShareAlike copyleft
t0 (P3/T0)	MultiNLI	MIT-style (NYU li-cense)	Yes	Permissive
t0 (P3/T0)	WinoGrande	CC BY 4.0 (da-taset)	Yes	Attribution required
t0 (P3/T0)	ANLI	CC BY-NC 4.0	No	Noncommercial re-striction
t0 (P3/T0)	SQuAD	CC BY-SA 4.0	No	ShareAlike copyleft
niv2 (Super-Natural In-structions v2)	NIv2 collection	Apache-2.0 (repo)	Partly	Collection is Apache-2.0; indi-vidual tasks may include content de-rived from up-stream datasets—check task cards
cot	GSM8K MIT	Yes	Permissive	
cot	AQuA-RAT	Apache-2.0	Yes	Permissive
cot	StrategyQA	MIT	Yes	Permissive
cot	QASC	CC BY 4.0	Yes	Attribution required
cot	e-SNLI MIT (repo); SNLI base	CC BY-SA 4.0	No	Underlying SNLI share-alike applies

cot	ECQA	CDLA-Sharing-1.0 (data)	No	Share-alike style data license; check obligations
cot	CREAK	MIT	Yes	Permissive
cot	QED	CC BY-SA 3.0 GFDL-derived	No	Wikipedia-derived; share-alike applies
cot	ComVE (Sense-Making)	CC BY-SA 4.0	No	ShareAlike copyleft
dialog	WikiDialog	CC BY-SA (reported)	No	Wikipedia-derived; share-alike applies
dialog	QReCC	CC BY-SA 3.0	No	ShareAlike copyleft
dialog	OR-QuAC	CC BY-SA 4.0	No	ShareAlike copyleft
dialog	QuAC MIT (per HF card);	CC BY-SA 4.0 on site	No	Conflicting sources; be conservative
dialog	Taskmaster-1	CC BY 4.0 (secondary sources)	Yes	Attribution required

3.2.3 Data Processing Pipeline

The data processing pipeline is essential for preparing diverse instruction-tuning datasets for language model fine-tuning. Raw datasets from different sources often have inconsistent formats, varied quality, and different conversation structures. Our pipeline standardizes these datasets into a unified format suitable for training conversational AI models, ensuring consistency while preserving the semantic content and task-specific information.

The pipeline transforms heterogeneous datasets into a standardized prompt-completion format that can be efficiently used with training frameworks like Hugging Face's TRL library.



The pipeline consists of five main steps:

1. **Download**: Fetch datasets from Hugging Face or other sources
2. **Standardize**: Convert to unified schema with consistent column names
3. **Filter**: Apply optional filtering criteria to remove unwanted samples
4. **Unroll**: Convert to conversational prompt-completion format
5. **Apply Template**: Format using chat templates for specific models

3.2.3.1 Download

Purpose: Fetch datasets from external sources, primarily Hugging Face Hub.

Implementation: The `download_datasets.py` module handles downloading datasets using the Hugging Face `datasets` library. Downloaded data is saved as Parquet files for efficient processing.

Sciriff Example:

```
{
  "input": "You will be presented with a citation segment from the section
of an NLP research paper, as well as the context surrounding that citation.
Classify the intent behind this citation by choosing from one of the follow
ing categories:\n- Background: provides context or foundational information
related to the topic.\n- Extends: builds upon the cited work.\n- Uses: appl
ies the methods or findings of the cited work.\n- Motivation: cites the wor
k as inspiration or rationale for the research.\n- CompareOrContrast: compa
res or contrasts the cited work with others.\n- FutureWork: cites the work
as a direction for future research.\n\nYour answer should be a single word
from the following list of options: [\"Background\", \"Extends\", \"Uses\",
\"Motivation\", \"CompareOrContrast\", \"FutureWork\"]. Do not include any
other text in your response.\n\nSection Title:\nintroduction\n\nContext bef
ore the citation:\nThus, over the past few years, along with advances in th
e use of learning and statistical methods for acquisition of full parsers (
Collins, 1997; Charniak, 1997a; Charniak, 1997b; Ratnaparkhi, 1997), signif
icant progress has been made...",
  "output": "Background",
  "metadata": {
    "domains": ["artificial_intelligence"],
    "input_context": "multiple_paragraphs",
    "output_context": "label",
    "source_type": "single_source",
    "task_family": "classification"
  },
  "_instance_id": "acl_arc_intent_classification:train:0"
}
```

3.2.3.2 Standardization

Purpose: Convert diverse dataset formats into a unified schema with consistent column names, data types, and task categories.

Schema: All datasets are standardized to this format:

- `instruction`: Task description or query (serves as system prompt) (string)
- `context`: The user's input (string)
- `response`: Expected output or answer (string)
- `task_category`: Type of task (e.g., "multiple_choice", "summarization")
- `source_dataset`: Original dataset name (string)
- `language`: Content language (string)
- `source_document_id`: Optional identifier for source tracking

Implementation: The `data_standard.py` contains dataset-specific standardizer classes. Each dataset has a custom `DatasetStandardizer` subclass that knows how to transform its specific format.

Sciriff Example:

```
{
  "instruction": "You are a helpful assistant. Answer the user's query.",
  "context": "You will be presented with a citation segment from the sectio
n of an NLP research paper, as well as the context surrounding that citatio
n. Classify the intent behind this citation by choosing from one of the fol
lowing categories:\n- Background: provides context or foundational informat
ion related to the topic.\n- Extends: builds upon the cited work.\n- Uses:
applies the methods or findings of the cited work.\n- Motivation: cites the
work as inspiration or rationale for the research.\n- CompareOrContrast: co
mpares or contrasts the cited work with others.\n- FutureWork: cites the wo
rk as a direction for future research.\n\nYour answer should be a single wo
rd from the following list of options: [\"Background\", \"Extends\", \"Uses
\", \"Motivation\", \"CompareOrContrast\", \"FutureWork\"]. Do not include
any other text in your response.\n\nSection Title:\nintroduction\n\nContext
before the citation:\nThus, over the past few years, along with advances in
the use of learning and statistical methods for acquisition of full parsers
(Collins, 1997; Charniak, 1997a; Charniak, 1997b; Ratnaparkhi, 1997), signi
ficant progress has been made...",
  "response": "Background",
  "task_category": "multiple_choice",
  "source_dataset": "sciriff",
  "language": "en",
  "source_document_id": "acl_arc_intent_classification:train:0"
}
```

3.2.3.3 Optional Filtering

Purpose: Remove samples based on specified criteria (e.g., language, length, quality).

Implementation: The `filter_dataset.py` module applies pandas-style filter expressions. If no filters are specified, this step copies the standardized file unchanged.

Example:

```
# Apply language filtering
filter_expressions = ["language == 'en'"]
# Apply length filtering
filter_expressions = ["length > 100", "task_category == 'multiple_choice'"]
```

3.2.3.4 Unrolling

Purpose: Transform standardized data into conversational prompt-completion format suitable for instruction fine-tuning, by converting instruction+context+response into a structured conversation format with roles and messages.

Implementation: Different unroll scripts handle different conversation types:

- `unroll_single_turn.py` - For single turn Q&A datasets (like sciriff)
- `unroll_multi_turn_oasst1.py` - For multi-turn conversations
- `unroll_gptnl_dataset.py` - For SPIN-specific formats

Sciriff Example:

```
{
  "task_category": "multiple_choice",
  "source_dataset": "sciriff",
  "language": "en",
  "message_tree_id": "a34de674-bbd6-4f51-b0d7-f5c93056b783",
  "row_id": "a34de674-bbd6-4f51-b0d7-f5c93056b783",
  "multi_turn": false,
  "prompt": [
    {
      "role": "user",
      "content": "You are a helpful assistant. Answer the user's query. You
will be presented with a citation segment from the section of an NLP resear
ch paper, as well as the context surrounding that citation. Classify the in
tent behind this citation by choosing from one of the following categories:
\n- Background: provides context or foundational information related to the
topic.\n- Extends: builds upon the cited work.\n- Uses: applies the methods
or findings of the cited work.\n- Motivation: cites the work as inspiration
or rationale for the research.\n- CompareOrContrast: compares or contrasts
the cited work with others.\n- FutureWork: cites the work as a direction fo
r future research.\n\nYour answer should be a single word from the followin
g list of options: [\"Background\", \"Extends\", \"Uses\", \"Motivation\",
\"CompareOrContrast\", \"FutureWork\"]. Do not include any other text in yo
ur response.\n\nSection Title:\nintroduction\n\nContext before the citation
:\nThus, over the past few years, along with advances in the use of learnin
g and statistical methods for acquisition of full parsers..."
    }
  ],
  "completion": [
    {
      "role": "assistant",
      "content": "Background"
    }
  ]
}
```

3.2.3.5 Template Application

Purpose: Apply model-specific chat templates to convert conversational format into final training strings.

Implementation: The `apply_template.py` module uses Hugging Face tokenizers to apply chat templates. Currently uses Gemma-3-4B-it template but can be configured for other models.

Chat Template: The pipeline uses templates that format conversations with special tokens:

- `<bos>` - Beginning of sequence
- `<start_of_turn>user` - User message start
- `<start_of_turn>model` - Assistant message start
- `<end_of_turn>` - Turn end marker

Sciriff Example:

```
{
  "task_category": "multiple_choice",
  "source_dataset": "sciriff",
  "language": "en",
  "message_tree_id": "a34de674-bbd6-4f51-b0d7-f5c93056b783",
  "row_id": "a34de674-bbd6-4f51-b0d7-f5c93056b783",
  "multi_turn": false,
  "prompt": "<bos><start_of_turn>user\nYou will be presented with a citatio
n segment from the section of an NLP research paper, as well as the context
surrounding that citation. Classify the intent behind this citation by choo
sing from one of the following categories:\n- Background: provides context
or foundational information related to the topic.\n- Extends: builds upon t
he cited work.\n- Uses: applies the methods or findings of the cited work.\
n- Motivation: cites the work as inspiration or rationale for the research.
\n- CompareOrContrast: compares or contrasts the cited work with others.\n-
FutureWork: cites the work as a direction for future research.\n\nYour answ
er should be a single word from the following list of options: [\"Backgroun
d\", \"Extends\", \"Uses\", \"Motivation\", \"CompareOrContrast\", \"Future
Work\"]. Do not include any other text in your response.\n\nSection Title:\
nintroduction\n\nContext before the citation:\nThus, over the past few year
s, along with advances in the use of learning and statistical methods for a
cquisition of full parsers...<end_of_turn>\n",
  "completion": "<start_of_turn>model\nBackground<end_of_turn>\n",
  "all": "<bos><start_of_turn>user\nYou will be presented with a citation s
egment from the section of an NLP research paper, as well as the context su
rrounding that citation. Classify the intent behind this citation by choosi
ng from one of the following categories:\n- Background: provides context or
foundational information related to the topic.\n- Extends: builds upon the
cited work.\n- Uses: applies the methods or findings of the cited work.\n-
Motivation: cites the work as inspiration or rationale for the research.\n-
CompareOrContrast: compares or contrasts the cited work with others.\n- Fut
ureWork: cites the work as a direction for future research.\n\nYour answer
should be a single word from the following list of options: [\"Background\"
, \"Extends\", \"Uses\", \"Motivation\", \"CompareOrContrast\", \"FutureWor
k\"]. Do not include any other text in your response.\n\nSection Title:\nin
troduction\n\nContext before the citation:\nThus, over the past few years,
along with advances in the use of learning and statistical methods for acqu
isition of full parsers...<end_of_turn>\n<start_of_turn>model\nBackground<e
nd_of_turn>\n"
}
```

3.2.3.6 Optional Task Inference

For datasets without explicit task categories (HuggingfaceH4Instruct and AyaDataset), the pipeline includes an optional task inference step using the `infer_task_types.py` module. This uses a language model to automatically classify samples into task categories:

- `qa_no_context`: Direct questions without additional context
- `qa_with_context`: Questions requiring provided context
- `summarization`: Text condensation tasks
- `multiple_choice`: Selection tasks
- `information_extraction`: Data extraction tasks
- `reasoning`: Logic and mathematical reasoning

The inference uses a Gemma model with carefully crafted prompts to ensure consistent classification.

3.2.3.7 Classification prompt

```
You are a task classification expert. Your job is to classify a text sample into ex
actly ONE of the following task types.

### Categories

- qa_no_context: A direct question with a definite answer that does NOT rely on add
itional provided context.
  Example: "What is the deepest abyss in the world?"
```

```
- qa_with_context: A direct question that CAN ONLY be answered using information pr
ovided in the prompt (a passage, list, or context section).
  Example: "What important event does the following text describe? This tiger has b
een struggling since 2004, as it lived primarily in Aceh, the northernmost tip of S
umatra. During the 2004 tsunami, much of this nature reserve was washed away. Vario
us organisations are now working to rebuild this nature reserve to ensure the survi
val of the Sumatran tiger."

- summarization: Condensing longer text into a shorter form.
  Example: "Summarize this 500-word article about climate change in 2-3 sentences."

- simplification: Rewriting complex information into simpler terms.
  Example: "Explain quantum physics in simple terms for a 10-year-old."

- multiple_choice: Multiple choice questions or selection tasks.
  Example: "Which of the following is NOT a mammal? A) Dog B) Cat C) Fish D) Whale"

- chat: Conversational or social dialogue.
  Example: "Hello! How are you doing today?" → "I'm doing well, thank you!"

- generation: Creative, open-ended content creation. **Not a direct question.** Som
etimes the TEXT ends with ...
  Example: "Write a short story about a dragon who loves to bake cookies."

- brainstorming: Generating multiple ideas or solutions.
  Example: "Give me 5 creative ideas for a company team-building event."

- reasoning: Logical reasoning, problem-solving, or analytical tasks.
  Example: "If all roses are flowers and some flowers are red, can we conclude that
some roses are red?"

- information_extraction: Extracting specific information from given text.
  Example: "Extract the names, dates, and locations mentioned in this news article.
"

### Decision Rules
1. If the TEXT is a **question**, classify it as QA (choose `qa_no_context` or `qa_
with_context`, never `generation`).
2. If the TEXT is shorter than 4 words, classify as generation.
3. Output ONLY the task type (e.g., `qa_no_context`). No explanation.
```

3.2.3.8 Goeievraag.nl data filtering with LLM-as-a-judge

The Goeievraag.nl dataset contains QA pairs from the goeievraag.nl website. This is a great source of Dutch data, but considering it is sourced from unmoderated user responses, it requires additional processing steps to ensure appropriate quality for instruction fine-tuning. To this extent we do the following:

1. Filter out unwanted topics
2. Only keep questions with a best answer
3. Apply PII (using the PrivateAI[2] tool) + toxic language detection
4. Use LLM to determine which QA pairs will make it to the final dataset

3.2.3.9 LLM-as-a-judge scoring prompt

```
## Task Introduction

You will evaluate question-answer pairs from an online Dutch forum to determine the
```

[2] https://www.privateai.com/

```
ir suitability for instruction fine-tuning an LLM. The goal is to identify high-qua
lity examples that teach a model to follow instructions accurately, provide helpful
responses, and avoid harmful biases. To do this, rate each QA pair across five crit
eria using a 1-5 scale.

## Evaluation Criteria

### 1. Instruction Following (1-5)
Evaluates how well the answer addresses what was specifically asked in the question
. Consider whether the response directly tackles the core request, stays on topic t
hroughout, and uses an appropriate format for the type of question asked. High scor
es indicate the answer comprehensively addresses all parts of the question without
unnecessary tangents.

### 2. Correctness & Accuracy (1-5)
Assesses the factual accuracy and reliability of the information provided. This inc
ludes checking whether claims are truthful, procedures are correct, reasoning is so
und, and any limitations or uncertainties are appropriately acknowledged. Consider
if the information is current and whether any advice could be potentially harmful i
f incorrect.

### 3. Helpfulness & Completeness (1-5)
Measures how useful the answer would be to someone with the original problem or que
stion. Evaluate whether the response provides sufficient detail to be actionable, i
ncludes important steps or considerations, and offers practical value. Consider if
key information is missing that would prevent the questioner from successfully appl
ying the answer.

### 4. Bias & Fairness (1-5)
Examines whether the response treats all people and groups respectfully and fairly.
Look for discriminatory language, harmful stereotypes, unfair assumptions about the
questioner, or biased perspectives on controversial topics. High scores indicate in
clusive language and balanced treatment of different viewpoints where appropriate.

### 5. Clarity & Communication (1-5)
Evaluates how well the answer is communicated and structured. Consider whether the
response is easy to understand, logically organized, uses appropriate language for
the context, and maintains a helpful and professional tone throughout. Assess if co
mplex concepts are explained clearly and the overall readability is good.

**Specific deductions for this criterion:**
- **Rate 1/5** if the answer mentions "GV", "Goeievraag" or synonyms
- **Lower rating** for answers written primarily in opinion form rather than inform
ative/instructional tone
- **Lower rating** for answers containing edit markers (e.g., "toegevoegd na [..]",
"EDIT:", etc.)
- **Lower rating** for answers that reference links or external sources without pro
viding actual URLs

## Evaluation Steps

### Step 1: Initial Reading
- Read the question and identify what is specifically being asked
- Read the entire answer and note its main approach

### Step 2: Score Each Criterion
For each of the 5 criteria:
1. **Apply the specific evaluation focus** (instruction following, accuracy, etc.)
2. **Use the 1-5 scale anchors** provided above

---
```

```
**Output Format:**
- Instruction Following: X/5 - [justification]
- Correctness: X/5 - [justification]
- Helpfulness: X/5 - [justification]
- Bias & Fairness: X/5 - [justification]
- Clarity: X/5 - [justification]
```

3.2.3.10 Results of LLM-as-a-judge scoring

After scoring the QA pairs marked with a best answer, we did qualitative analysis to pick a strict threshold of total score >= 20 and score of each category >= 4. After applying this filter, the dataset size decreased from ~62k to 19,778 samples.

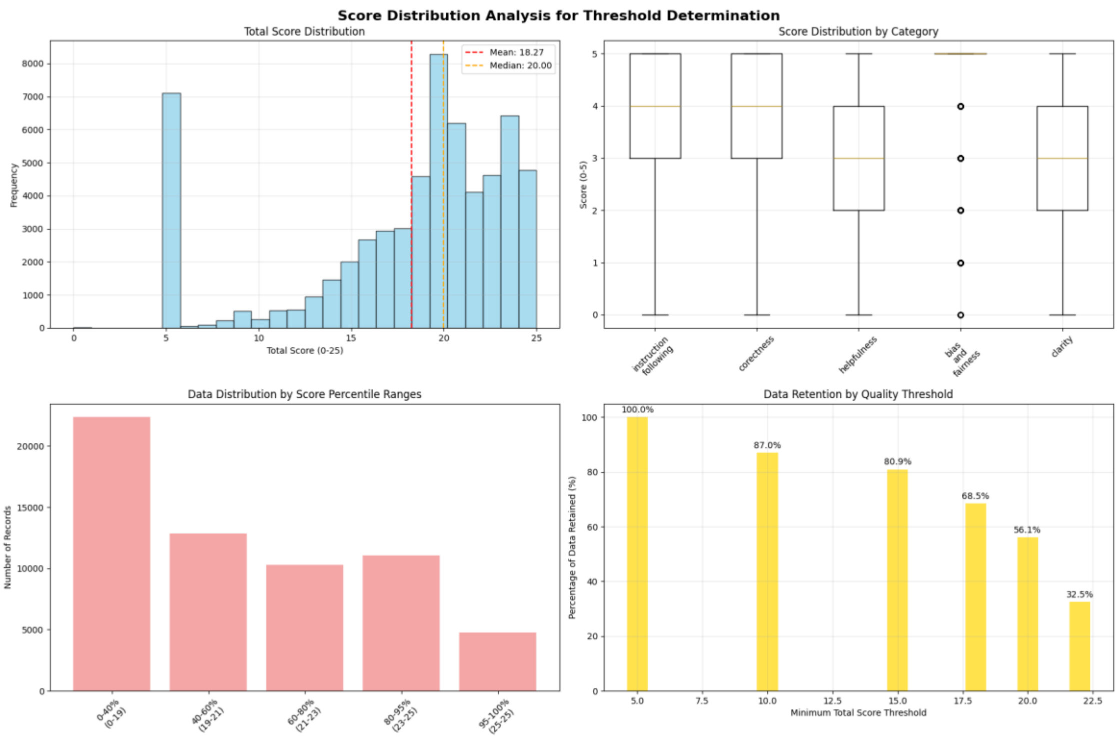


Figure 13: Goievraag evaluation plots

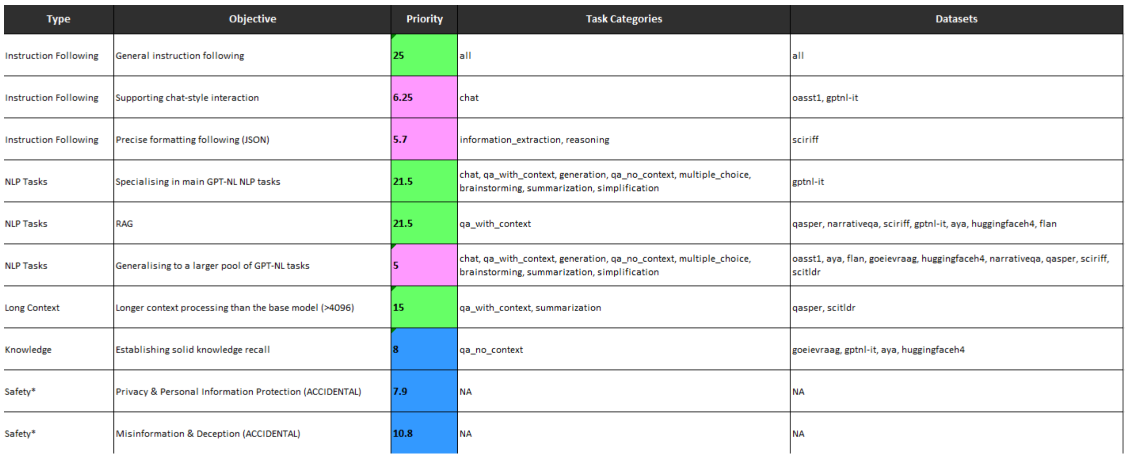
3.3 Data selection process

Data selection for instruction-tuning LLMs focuses on identifying and curating high-quality training examples from larger datasets. Rather than using all available instruction-response pairs indiscriminately, data selection employs various filtering and ranking strategies to identify the most valuable examples for model training. Data selection serves two primary objectives that are fundamental to developing effective instruction-tuned models:

1. It acts as a quality filter, systematically removing low-quality examples that could degrade model performance. This includes filtering out responses that are factually incorrect, poorly written, off-topic or contain harmful content.

2. Data selection enables alignment with specific priorities and objectives defined for the model's intended use cases.

It is desired that the instruction-tuned GPT-NL checkpoints must exhibit the following objectives, accompanied by a priority score:



Type	Objective	Priority	Task Categories	Datasets
Instruction Following	General instruction following	25	all	all
Instruction Following	Supporting chat-style interaction	6.25	chat	oasst1, gptnl-it
Instruction Following	Precise formatting following (JSON)	5.7	information_extraction, reasoning	sciriff
NLP Tasks	Specialising in main GPT-NL NLP tasks	21.5	chat, qa_with_context, generation, qa_no_context, multiple_choice, brainstorming, summarization, simplification	gptnl-it
NLP Tasks	RAG	21.5	qa_with_context	qasper, narrativeqa, sciriff, gptnl-it, aya, huggingfaceh4, flan
NLP Tasks	Generalising to a larger pool of GPT-NL tasks	5	chat, qa_with_context, generation, qa_no_context, multiple_choice, brainstorming, summarization, simplification	oasst1, aya, flan, goeievraag, huggingfaceh4, narrativeqa, qasper, sciriff, scitldr
Long Context	Longer context processing than the base model (>4096)	15	qa_with_context, summarization	qasper, scitldr
Knowledge	Establishing solid knowledge recall	8	qa_no_context	goeievraag, gptnl-it, aya, huggingfaceh4
Safety*	Privacy & Personal Information Protection (ACCIDENTAL)	7.9	NA	NA
Safety*	Misinformation & Deception (ACCIDENTAL)	10.8	NA	NA

GPT-NL Priorities

These priorities highlight the need for a data selection method that can balance multiple objectives while scaling to large dataset sizes efficiently.

The challenge of identifying the most useful training samples for instruction fine-tuning is complex, as there is no straightforward method to determine which examples will contribute most effectively to model performance. The quality of instruction-response pairs can depend on numerous factors including linguistic complexity, task diversity, response accuracy, alignment with target capabilities and subtle patterns that may not be immediately apparent through manual inspection. This leads us to experiment with three methodologies developed in the academic world.

3.3.1 Representation-based Data Selection Plus (RDS+)

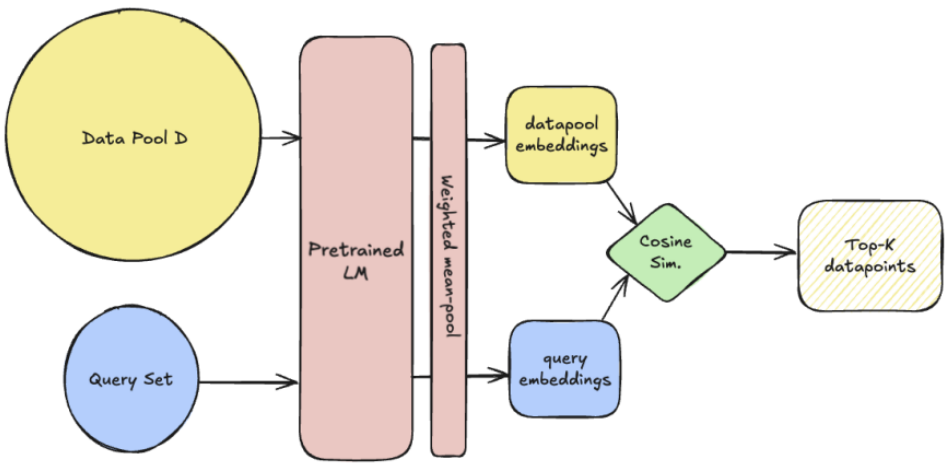


Figure 14: RDS+ Workflow

The core insight behind RDS+ is that the hidden representations from a pre-trained model's last layer capture semantic similarity better than dedicated embedding models or gradient-based approaches. By using these representations to compute cosine similarity between query samples and candidate training data, RDS+ can efficiently identify the most relevant training examples.

Our implementation strategy leverages RDS+ in a systematic pipeline designed to maximize alignment with GPT-NL priorities:

1. Priority-Aligned Test Set Curation

We begin by manually curating a comprehensive test set that directly reflects our priority matrix, across task types and source datasets.

2. Query Set Creation

From our curated test set, we split a validation subset to serve as RDS+ queries. This way, the validation split represents the testing set distribution without causing data leakage.

3. Training Data Subsampling

Using RDS+, we subsample our large training corpus by computing similarity scores between each training example and our validation queries. The round-robin selection algorithm ensures balanced representation across all priority areas while identifying the most relevant training samples.

3.3.2 G-Eval: LLM-as-a-judge filtering

G-Eval leverages LLMs as judges to assess the quality of instruction-response pairs. Rather than relying on simple metrics or heuristics, G-Eval uses LLMs to provide nuanced quality assessments that closely mirror human judgment. The methodology works by presenting a LLM with detailed evaluation criteria and asking it to score examples across multiple quality definitions. The LLM judge examines each instruction-response pair and assigns numerical scores based on carefully crafted rubrics that define what constitutes quality at different levels.

For our data selection process, we employed five distinct evaluation rubrics, each targeting a critical aspect of training data quality:

- **Language Quality** evaluates whether the prompt-completion pair maintains consistent language use (English or Dutch) without inappropriate mixing or code-switching. This ensures our training data maintains linguistic coherence.
- **Prompt Completeness** assesses whether the instruction is clear, unambiguous, and provides sufficient context for understanding what the response should contain. Well-defined prompts are essential for effective instruction fine-tuning.
- **Completion Helpfulness** measures how well the response addresses the prompt while remaining concise and relevant. This rubric filter out responses that are off-topic, repetitive, or unnecessarily verbose.
- **Completion Truthfulness** evaluates factual accuracy and ensures responses don't contain hallucinations or invented information not present in the provided context or general knowledge.
- **Harmlessness** ensures the content is safe, respectful, and free from harmful, discriminatory, or inappropriate material that could pose risks to various forms of welfare.

Each rubric uses a 5-point scale, with detailed descriptions for each score level to ensure consistent evaluation. To maintain quality standards in our final dataset, we established a threshold of 3 or higher across all five rubrics - meaning samples must achieve at least a “moderately acceptable” rating in every dimension to be included in our training set.

3.3.3 UltraFineWeb FastText classifier

The Ultra-FineWeb classifier represents a lightweight approach to data quality assessment that leverages fastText for rapid content evaluation. Unlike the sophisticated LLM-based evaluation methods, this classifier prioritizes computational efficiency while maintaining effective filtering capabilities, making it particularly suitable for processing large-scale datasets where inference speed is crucial.

The methodology behind the Ultra-FineWeb classifier involves training a fastText model to distinguish between high- and low-quality text samples using carefully curated seed data. Although the Ultra-FineWeb classifier was originally designed and optimised for pre-training data filtering, we decided to explore its utility for instruction fine-tuning data selection. Our hypothesis was that this classifier could serve as an effective filter to identify and remove outright poorly formatted data or content with significant grammatical errors that would be detrimental to instruction fine-tuning. Pre-training data and instruction fine-tuning data share certain fundamental quality characteristics: both benefit from proper formatting, grammatical correctness, and linguistic coherence, making this cross-domain application a reasonable experimental approach.

The appeal of using this pre-trained classifier lies in its ability to quickly process large volumes of instruction-response pairs and flag obviously problematic content such as garbled text, severe formatting issues, or content with substantial linguistic errors. While it may not capture the nuanced quality aspects specific to instruction-following tasks, it can efficiently eliminate the obviously bad quality samples, allowing more sophisticated evaluation methods to focus on the remaining, higher-quality candidates.

3.3.4 Conclusion: Comparing Data Selection Approaches

3.3.4.1 Individual Method Impact Analysis

Each of the three data selection methods demonstrated distinct filtering patterns across task categories and source datasets, revealing different biases and strengths in their quality assessment approaches.

RDS+

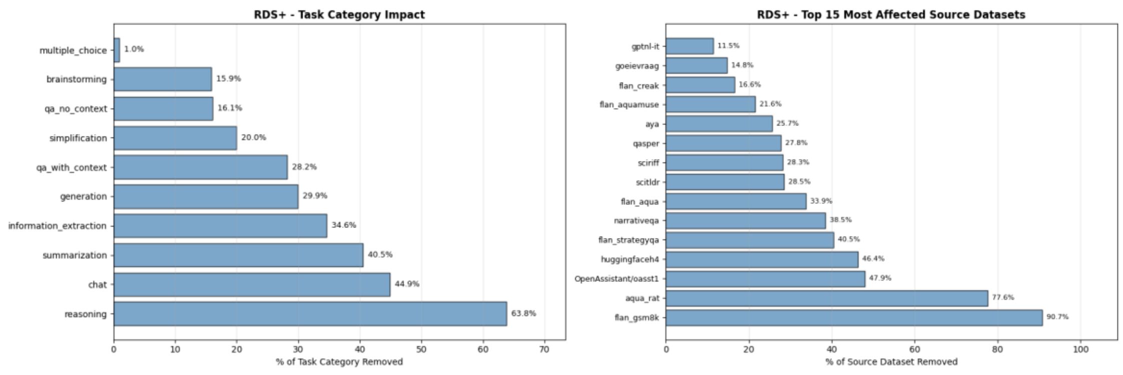


Figure 15: RDS+ Filter Impact

RDS+ demonstrated a balanced filtering strategy, with the highest removal rate for reasoning tasks (63.8%) and moderate filtering across most categories. The method showed relatively uniform impact across source datasets, with the most affected being flan_gsm8k (90.7%) and aqua_rat (77.6%). This suggests RDS+ focuses on semantic relevance to the target priority distribution rather than absolute quality, making it conceptually different from the other two filtering approaches.

G-Eval: LLM-as-a-judge Filtering

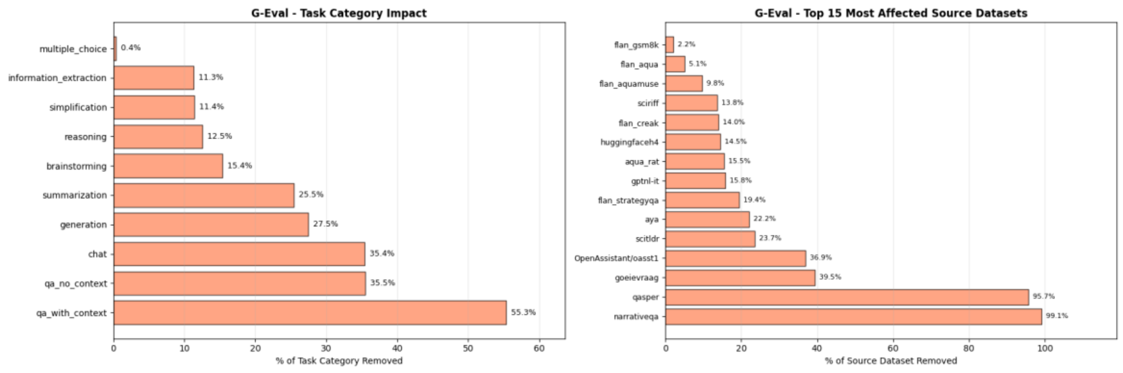


Figure 16: G-Eval Filter Impact

G-Eval's filtering showed a strong focus on context-dependent tasks, removing 55.3% of qa_with_context samples while being more lenient with simpler tasks like multiple_choice (0.4% removed). The method heavily filtered specific datasets like narrative Q&A (99.1%) and qasper (95.7%), suggesting these sources contained responses that failed to meet the stringent rubric requirements for truthfulness, helpfulness, or completeness. This pattern indicates G-Eval's strength in identifying quality issues in long-form, context-heavy responses but potentially being overly conservative.

FastText Classifier

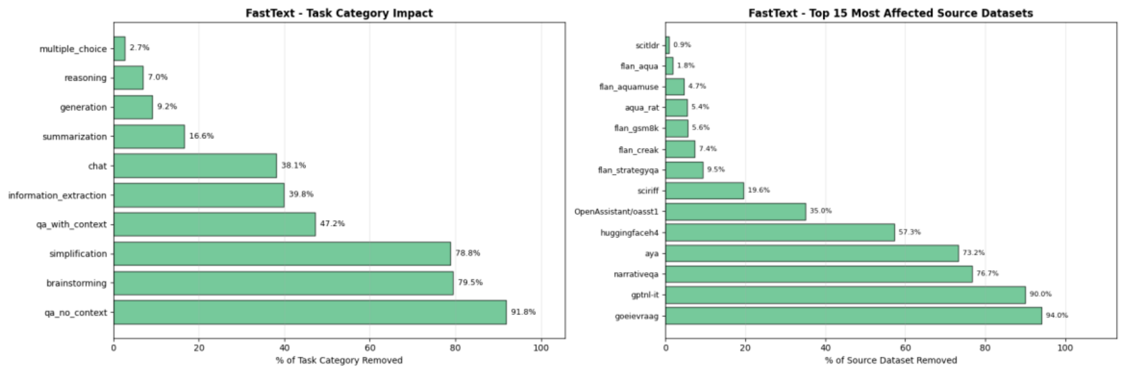


Figure 17: FastText filter Impact

FastText Filter Impact

The FastText classifier mostly removed samples containing Dutch samples. This is likely a consequence of the classifier not being trained on Dutch samples at all. This approach appears to prioritise surface-level quality indicators like grammar and structure over semantic appropriateness.

3.3.4.2 Agreement Between Methods

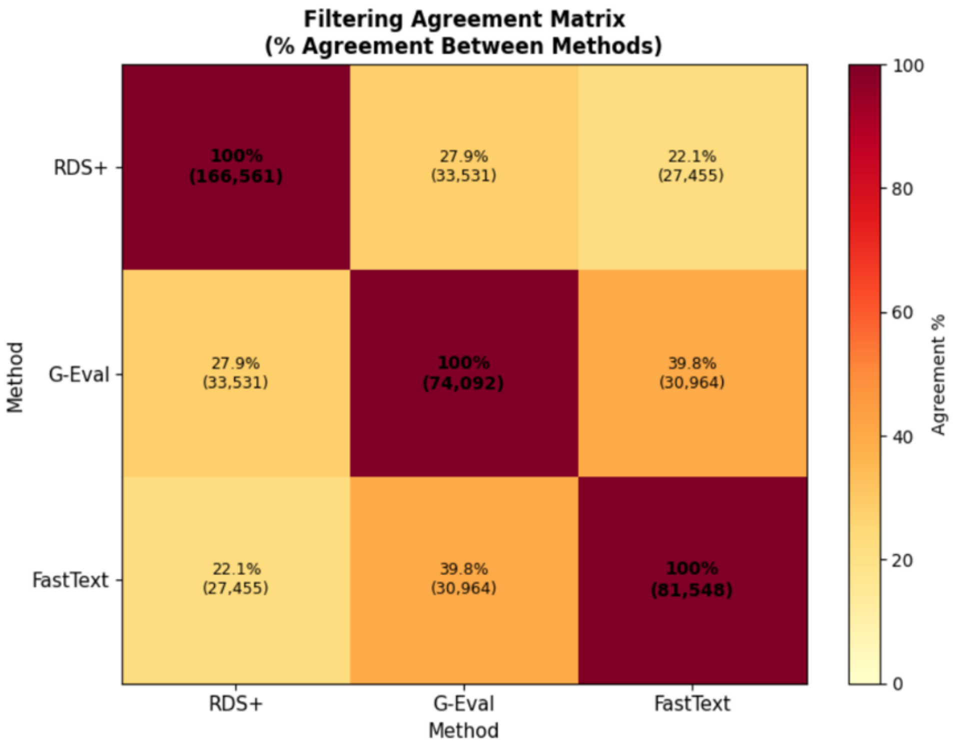


Figure 18: Filtering Agreement Matrix

The agreement analysis between the three methods reveals a fundamental challenge in data quality assessment: **quality is inherently difficult to estimate**, and different methods capture different aspects of what constitutes valuable training data. The agreement matrix shows:

- **RDS+ and G-Eval**: Only 27.9% agreement (33,531 samples)
- **RDS+ and FastText**: Only 22.1% agreement (27,455 samples)
- **G-Eval and FastText**: 39.8% agreement (30,964 samples)

These low agreement rates indicate that each method operates on fundamentally different principles. RDS+ prioritizes semantic relevance to target capabilities, G-Eval focuses on multi-dimensional quality rubrics, and FastText emphasizes linguistic and formatting correctness. The lack of consensus suggests that “quality” in instruction fine-tuning data is multi-faceted and no single method captures all relevant dimensions.

3.3.4.3 Impact on Model Performance

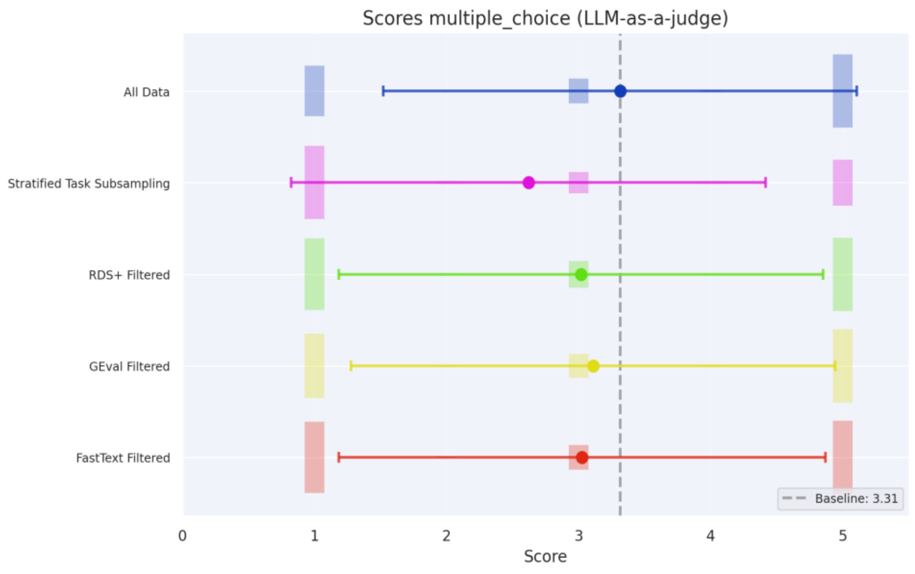


Figure 19: Filtering results comparison

Perhaps most surprisingly, **none of the filtering solutions improved upon the baseline of using all available data.**

All three filtering approaches, along with stratified subsampling, showed performance below the baseline. This counterintuitive result can likely be attributed to our **low data setting**. With limited training data available, aggressive filtering may remove examples that, despite quality concerns, still contribute valuable signal for learning instruction-following behavior. In resource-constrained scenarios, the diversity and volume of training data may matter more than individual example quality.

3.4 Configuring, Running and Monitoring the Instruction Fine-Tuning

This page provides a comprehensive overview of the Instruction Fine-Tuning process for GPT-NL, covering execution frameworks, configuration, training, and monitoring. We have already described how the data is prepared and selected. Here, we focus on the implementation, workflow, and experiments for supervised fine-tuning (SFT) on our HPC infrastructure.

3.4.1 Frameworks Choice

After analysing multiple frameworks for instruction fine-tuning, including HuggingFace TRL, ColossalAI, DeepSpeedChat, Open-Instruct, and others, we chose HuggingFace TRL as our **primary fine-tuning framework**, augmented with:

- **DeepSpeed ZeRO** for multi-GPU/multi-node scaling (Stages 1–3).
- **Dataset streaming** (for memory efficiency on large parquet datasets).
- **Synchronization primitives** (avoid races in model/dataset caching across many ranks).
- **Custom callbacks** for **throughput** and **sample generation** into monitoring tools.

Detailed tables, comparisons of frameworks, RLHF methods have been moved to the Appendix **Fine-tuning frameworks** for clarity.

Why TRL for SFT

- Wide adoption in the research and open-source community
- Extensive documentation and examples
- Support for SFT, DPO, PPO, and full finetuning
- **Dataset format flexibility**: prompt-completion (string or conversational) and language-modeling style with data collators
- Deep integration with **Accelerate** and **DeepSpeed**
- Active ecosystem: ready-to-use trainers (`SFTTrainer`) and configs (`SFTConfig`)

*The complete comparison of frameworks and RLHF approaches is available in the Appendix **Fine-tuning frameworks**.*

3.4.2 Configuration

Proper configuration of a fine-tuning run is critical to ensure efficient resource utilization and optimal model performance. In this section, we describe the essential steps required to initiate a fine-tuning process—specifically, how to configure Supervised Fine-Tuning (SFT).

Setting up an SFT process involves defining several key options:

- Model Selection: Specify which pre-trained model will be fine-tuned. This can be a local model checkpoint (e.g., a path to the model at a specific epoch) or a publicly available model for benchmarking and comparison.
- Dataset Choice: Determine the dataset to use for training. As outlined in the Data preparation and Data selection sections, options include internal datasets such as SPIN and selected public datasets.
- Fine-Tuning Configuration: Define how the fine-tuning should proceed, including training duration, hyperparameters (e.g., learning rate, batch size), and allocation of HPC resources (e.g., number of GPUs, memory requirements).

3.4.2.1 YAML Configuration file

We try to make these options explicit and easy to configure by using YAML configuration files. We have a folder named `config` where we hierarchically store the configurations for our runs. The most important parameters are:

- `model_name_or_path`: the input model to finetune
- `dataset_name_or_path`: the dataset to use. Currently supported: parquet files with a supported type of dataset by TRL. It can be a single file or multiple files (if you use a `*` in the variable). If a single file is used, it must contain a column named `split` to indicate which rows are for training and which ones for testing. If a wildcard (`*`) is used, train and test splits are loaded separately: this has been implemented to be able to stream datasets instead of loading them as a bulk in the beginning of the training. See below the streaming datasets section for more details.
- `output_dir`: where to save the finetuned model
- `chat_template_path`: to configure a specific chat template for finetuning the model

- `DeepSpeed`: path to a DeepSpeed configuration file, useful to choose the distributed strategy (stage 1 / 2 / 3)
- **Model**
  - `model_name_or_path`: the input model to finetune (HF hub or local path to checkpoint)
  - `tokenizer_name_or_path` (optional; defaults to `model_name_or_path`)
  - `trust_remote_code` (for custom model code)
  - **Long context** (optional): `rope_scaling_type`, `rope_scaling_factor`
  - Attention backend: enable **FlashAttention2** or **SDPA** (LLaMA)
- **Dataset**
  - `dataset_name_or_path`: the dataset to use. Currently supported: parquet files with a supported type of dataset by TRL.
    - If using a **single file**, ensure it contains a `split` column with values `train|val|test`.
    - If using * (wildcard), provide `..._train.parquet`, `..._val.parquet`, optionally `..._test.parquet` to enable **streaming** (reduces memory required instead of loading the whole dataset in memory). See below the streaming datasets section for more details.
  - (Optional) `max_samples` and `validation_split_percentage`
- **Training & Output**
  - `output_dir`: where to save the finetuned model
  - `num_train_epochs`, `per_device_train_batch_size`, `gradient_accumulation_steps`
  - `learning_rate`, `lr_scheduler_type`, `warmup_steps`, `weight_decay`
  - Precision: `bf16|fp16|tf32`, `gradient_checkpointing`
  - `save_strategy`, `save_steps`, `save_total_limit`, `load_best_model_at_end`
  - `chat_template_path`: to configure a specific chat template for finetuning the model (must match inference)
- **Distributed**
  - `DeepSpeed`: path to ZeRO config (`ds_config_zero[1|2|3].json`)
  - Optional offloading: `offload_folder`, `offload_state_dict`
  - `low_cpu_mem_usage: true` to reduce load on memory

Example (condensed)

```yaml
# config/train/sft_config.yaml
model_name_or_path: PATH_TO_MODEL
tokenizer_name_or_path: PATH_TO_MODEL
trust_remote_code: true

# Long context (optional)
rope_scaling_type: dynamic       # or 'linear'
rope_scaling_factor: 2.0

# Attention backend
use_flash_attention_2: true      # or sdpa for LLaMA

# Dataset: stream by providing separate parquet files
```

```yaml
dataset_name_or_path: PATH_TO_DATASET_*.parquet

# Output & logging
output_dir: /output/gptnl-sft
report_to: ["wandb"]
chat_template_path: config/evaluate/chat_template_empty.jinja

# Training
num_train_epochs: 1
per_device_train_batch_size: 4
gradient_accumulation_steps: 4
learning_rate: 2.0e-5
lr_scheduler_type: cosine
warmup_steps: 500
weight_decay: 0.0
bf16: true
tf32: true
gradient_checkpointing: true

# Save/Eval
do_eval: true
eval_strategy: steps
eval_steps: 500
save_strategy: steps
save_steps: 500
save_total_limit: 3
load_best_model_at_end: false

# Distributed
DeepSpeed: config/train/ds_config_zero3.json
low_cpu_mem_usage: true
offload_folder: /scratch/offload
disable_cache: true
```

***Tip:** With **ZeRO-3**, memory footprint per GPU drops significantly, allowing **larger batches** and improved **tokens/sec**.*

3.4.2.2 Job Scripts and Execution Flow

The job scripts are stored in the `jobs` folder, and they contain job requirements specifications (resources):

- amount of nodes
- type of nodes
- usage of reservation
- duration of the job

These options cannot be easily moved to YAML configuration, so they need to be checked before usage in the corresponding job file. They are written as `#SBATCH` headers.

The rest of job scripts contain module loading and preparation of the training run (setting environment variables, loading the proper configuration file).

It is important when you create or use a job script that you check:

- which configuration file it loads: `TRAIN_CONFIG` env variable exported
- `#SBATCH` headers for resources

To submit a job to the queue, just use `sbatch jobs/<PATH_TO_JOB_SCRIPT>`.

3.4.3 Training

This section describes how the training works. It is classical supervised training, where there are input variables (chat conversations up to a specific point) and target variables (next assistant message). The learning goal is to minimize the loss on the target assistant message. In other words, the model learns to reply as the assistant (the target variable).

If we follow what happens when a job script is launched:

5. The job is granted resources, and SLURM runs a single copy of the batch script on the first node in the set of allocated nodes.
6. The batch script is executed: the environment is configured (modules are loaded, environment variables are set, including the `TRAIN_CONFIG`)
7. `srun` launches the tasks in all the configured nodes. In our setting, we have a task for each GPU. For example, if we run a fine-tuning on 2 nodes with 4 GPUs each, we will have 8 tasks running
8. `torchrun_launcher.sh` is executed: each task executes `torchrun` with target the `TRAINING_COMMAND` (defined in the job script) and connects to the master process using NCCL

Then the `TRAINING_COMMAND` is actually the python script that gets executed in parallel via `torchrun`. Its entry point is `src/train/train.py` where the real training happens:

- data loading: from parquet files, the data is formatted according to chat_template and tokenized
- model loading: using `HuggingFace` classes, the model and the tokenizer are loaded according to configuration
- DeepSpeed is getting configured and distributes the model accordingly to its stage
- batches start to be processed (this happens a few minutes later) and loss is computed for backpropagation
- checkpoints are saved

3.4.3.1 TRL classes

- `SFTConfig` – defines training parameters, optimizer, precision, checkpointing, DeepSpeed config, chat template, logging
- `SFTTrainer` – executes the training loop

3.4.3.2 DeepSpeed configuration

As documented in the pre-training documentation, we use DeepSpeed to distribute the training. For finetuning we tested the stage 1, 2 and 3.

3.4.3.3 DeepSpeed and memory constraints

DeepSpeed is strongly linked to the memory constraints. Stages 1 and 2 already distribute the optimizer (stage 1) and gradient (stage 2), but still the memory requirements are almost identical as DPP strategies, because the largest amount of memory is actually occupied by the model parameters (27B).

Stage 3, on the other side, partitions the parameters across multiple workers/devices and really enables to reduce the memory requirements of a single device, leaving room for more memory for larger batches. The slight performance decrease of stage 3 is totally and over-compensated by the speed of larger batches, that indeed would crash with stage 1 and 2.

We rely on **DeepSpeed ZeRO** sharding:

- **ZeRO-1**: Shards **optimizer states**. Memory relief is limited for very large models.
- **ZeRO-2**: Shards **gradients** as well. Still heavy if parameters stay replicated.
- **ZeRO-3**: Shards **parameters, gradients, and optimizer**. This is the **recommended** mode for **27B+** models; it enables **larger batch sizes** and **higher throughput**.

Important

- Do **not** use `device_map="auto"` with ZeRO/torchrun. We explicitly unset this in distributed mode to avoid incorrect HF auto-sharding.

DeepSpeed JSON highlights (example fields to inspect in `ds_config_zero3.json`)

```json
{
  "zero_optimization": {
    "stage": 3,
    "offload_param": {
      "device": "cpu",
      "pin_memory": true
    },
    "offload_optimizer": {
      "device": "cpu",
      "pin_memory": true
    }
  },
  "bf16": { "enabled": true },
  "gradient_accumulation_steps": 4,
  "train_micro_batch_size_per_gpu": 4
}
```

Memory & performance

- ZeRO-3 usually unlocks **per_device_train_batch_size ≥ 8** (depending on sequence length and model size).
- With dataset streaming, we observed **substantial throughput gains** (see Appendix B).

3.4.3.4 Chat_template

We can configure the specific `chat_template` used for fine-tuning. The pre-trained model has no `chat_template`. It is important that the same `chat_template` is used for finetuning and inference, as it is used for priming the model to respond.

3.4.3.5 Data loading

The training script loads the data from parquet files, according to the configuration.

TRL works out of the box with the following formats for SFT:

- Standard Prompt-Completion: prompt and completion are strings. Strings should already be formatted according to a `chat_template`

- conversational Prompt-Completion: prompt and completion are list of messages with `{role,content}` attributes. This is useful if we want to apply a different `chat_template` at runtime
- language-modelling (conversational) format: we don't need to manually prepare prompts and completions, with a simple `DataCollator` configuration we can tell TRL to use as targets all the `assistant` messages (intermediate and final ones).

TRL automatically detects the type of the dataset passed and whether it needs to be tokenized (strings) or not (token_ids) and configures the batches to be provided to the different workers in a distributed setting.

3.4.3.6 Additions on top of TRL

Synchronization primitives

Some actions make the jobs fail when more than 8 workers are in place (probability of race conditions increases):

- downloading models (when not local model or not yet downloaded in cache)
- creating cache of datasets (when loading dataset)

For these cases, we used barriers:

```
if is_main_process():
    # main process loads first
    full_dataset = load_dataset(...)
# Wait for main process to finish downloading/caching
torch.distributed.barrier()
if not is_main_process():
    # then the other processes can load from cache without concurrent write
s
    full_dataset = load_dataset(...)
```

Streaming datasets

Easier dataset loading: load everything, transform it to tokenized, then efficiently dispatch sub-batches to workers.

Problem: large datasets do not fit in memory, so the processes crash

Solution: stream the dataset from the disk (parquet). Complications:

- The train and test datasets need to be separate on disk. This is why in the YAML configuration you see that we use the wildcard `*` to denote when the two files are separate. In this way it's possible to stream them independently.
- The total size is unknown in streaming mode, need to compute it first. This is fixed by doing a first iteration on the dataset.
- Global batch created in main process, and split for all the workers, `train_batch_size` needs to be adjusted to the global batch size

3.4.4 Monitoring

Monitoring uses TRL integration with **Weights & Biases**. We log:

- Configuration and hyperparameters

- Training metrics: loss, token accuracy, learning rate, steps, number of tokens processed
- System metrics: GPU usage, network, throughput per device and globally
- Example outputs during training via a custom `LLMSampleCallback` (text output by the model)

3.5 Instruction fine-tuning evaluation

Evaluating instruction-tuned models requires a multi-faceted approach to assessing both overall performance and task-specific capabilities. This document describes our evaluation methodology and presents initial results from fine-tuning experiments. We implement evaluation for two primary purposes:

1. **Comparing GPT-NL to third party models**: Benchmarking our models against state-of-the-art instruction-tuned models to understand relative performance on common task categories (linguistic understanding, reasoning, knowledge, etc.).
2. **Gathering insights into how we perform on the GPT-NL instruction fine-tuning priorities**: Understanding model performance across different task categories with more granular and deeper insights than average scores of metrics. We aim to understand to be able to adapt our fine-tuning approach accordingly.

To achieve these goals, we employ two complementary types of evaluation:

1. **Benchmark evaluation (EuroEval)**: Using standardized benchmarks for external comparison
2. **Internal test set evaluation**: Using task-categorized test sets with multiple evaluation approaches for deeper insights:
   - Traditional metrics (BLEU, ROUGE, METEOR)
   - Model-based metrics (BERTScore)
   - LLM-as-a-judge evaluation (Prometheus and GEval)

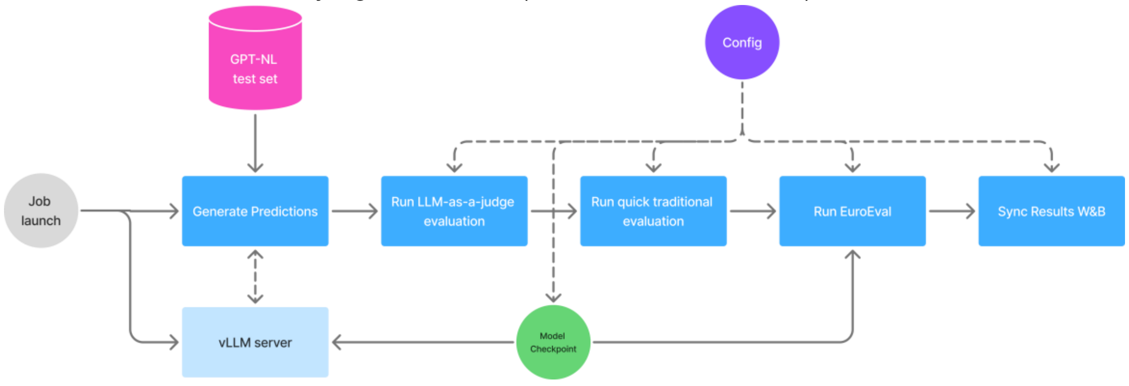


Figure 20: Instruction fine-tuning evaluation

The evaluation pipeline orchestrates multiple evaluation approaches in a single compute job with sequential stages. Starting with prediction generation using a vLLM server, the cached predictions feed into three parallel evaluation tracks: LLM-as-a-judge assessment, traditional metrics computation, and EuroEval benchmarking. All results are aggregated and synchronized to Weights & Biases for experiment tracking and analysis. We discuss these methods in the following.

3.5.1 Benchmark Evaluation: EuroEval

GPT-NL uses EuroEval as the core benchmark collection for evaluating instruction-tuned models. EuroEval provides standardized task benchmarks for both English and Dutch, enabling comparison with external models and tracking progress across model iterations. WP21 extends this collection with Dutch sets that are focused on Dutch language and culture, but that effort is described separately and here we consider the base sets available.

The implementation is re-used from the pre-training pipeline and is explained in this section.

3.5.2 Internal Test Set Evaluation

Our internal evaluation pipeline provides detailed insights into model performance across different task categories, enabling us to understand where the model excels and where improvements are needed. The pipeline follows a modular design with three sequential stages: prediction generation, traditional metrics computation, and LLM-as-a-judge evaluation. Predictions are generated once and cached for reuse, while results from all stages are aggregated and logged to Weights & Biases.

Our test set is constructed from a subset of the training datasets, carefully selected and categorized to ensure comprehensive coverage of different task types. The test set follows a standardized Parquet format with the following schema:

Column	Type	Description
`prompt`	str/list	The input prompt (string or conversational format as list of messages dict)
`completion`	str	The reference/ground truth response
`instruction`	str	The original instruction (before chat template application)
`task_category`	str	The task type for granular analysis
`source_dataset`	str	Original dataset name for tracking data provenance
`language`	str	Language code (‘en’ for English, ‘nl’ for Dutch)

Task categories enable granular performance analysis across different types of instructions:

Task Category	Description
`qa_with_context`	Question answering using provided context
`qa_no_context`	Question answering using general knowledge
`summarization`	Text condensation and summarization
`reasoning`	Mathematical and logical reasoning
`information_extraction`	Structured information retrieval
`generation`	Creative text generation
`simplification`	Text simplification for accessibility
`multiple_choice`	Selection of predefined options
`chat`	Multi-turn conversational dialogue
`brainstorming`	Idea generation and exploration
`dutch`	Dutch-specific language tasks

The test set is typically limited to around 5K samples to enable fast iteration during model development while maintaining coverage across all task categories.

Evaluation Criteria

The evaluation system is designed to assess multiple dimensions of response quality:

Generic Criteria (all tasks):

- **Helpfulness**: Does the response satisfy the user's intent in a complete, relevant and conciseness manner?
- **Truthfulness**: Is the response factual and free from hallucinations? When a piece of contextual information is given, the model can only use that to base its answer on, besides general knowledge.
- **Harmlessness**: Is the response harmless, respectful, and appropriate?
- **Language Quality**: Is the response accessible, fluent and with correct grammar? (Dutch or English specific)

Task-Specific Criteria:

- **Summarization**: Faithfulness, coverage, and conciseness
- **Simplification**: Meaning preservation and clarity, not summarized
- **QA with Context**: Accuracy using only provided context
- **QA without Context**: Appropriate use of general knowledge
- **Generation**: Creativity and engagement
- **Brainstorming**: Diversity and directional spread of ideas
- **Reasoning**: Logical structure and correctness
- **Chat**: Appropriate conversational flow
- **Information Extraction**: Exactness and completeness
- **Multiple Choice**: Correctness

3.5.2.1 Stage 1: Prediction Generation

Model predictions are generated using a **vLLM server** infrastructure for efficient batched inference. This approach enables:

- **High-throughput generation**: Optimized inference for large test sets
- **Format flexibility**: Support for both string prompts and conversational message formats
- **Infrastructure reuse**: Same deployment for training-time evaluation and post-training assessment

The generation process applies appropriate chat templates to prompts, uses configurable sampling parameters (temperature, top-p, repetition penalty), and produces responses that are cached for subsequent metric computation stages. Predictions are stored alongside metadata including task categories, languages, source datasets, and reference answers, enabling granular analysis across different data dimensions.

3.5.2.2 Stage 2: Traditional Metrics

Traditional metrics provide fast, reference-based evaluation of surface-level similarity and semantic alignment between predictions and references.

Metric Categories

The evaluation computes several complementary types of metrics:

Metric Category	Examples	Measures
Token overlap	METEOR	Synonym-aware word matching with stemming
N-gram recall	ROUGE-1, ROUGE-2, ROUGE-L	Unigram, bigram, and longest subsequence overlap
N-gram precision	BLEU	Corpus-level precision with brevity penalty
Semantic similarity	BERTScore	Contextual embedding alignment using multilingual DeBERTa

BLEU Aggregation: Unlike sample-level metrics, BLEU is computed at group level (combinations of task category, dataset, and language) as it requires corpus-level aggregation for meaningful interpretation.

BERTScore: Uses `microsoft/mdeberta-v3-base` to compute semantic similarity beyond exact word matching, capturing meaning alignment even when phrasing differs.

Multi-Level Aggregation

Metrics are aggregated across multiple dimensions to provide comprehensive analysis:

- **Overall**: Performance across the entire test set
- **By task category**: Identifying strengths in specific instruction types (QA, summarization, etc.)
- **By dataset**: Understanding which training data sources contribute to capabilities
- **By language**: Comparing Dutch vs English performance
- **By combinations**: Cross-tabulated analysis (e.g., Dutch summarization vs English summarization)

This multi-dimensional view enables identifying both broad patterns and specific areas needing improvement.

3.5.2.3 Stage 3: LLM-as-a-Judge Evaluation

Traditional metrics capture surface-level similarity but may miss nuanced aspects of response quality. GPT-NL implements **LLM-as-a-Judge** evaluation using two complementary approaches:

3. **Multilingual-Prometheus (M-Prometheus)** (Pombal et al., 2025): Specialized open-weight evaluation models (3B-14B parameters) extending the original Prometheus framework (Kim et al., 2024) to support multilingual assessment across 20+ languages through direct assessment and pairwise comparison. Prometheus models are specifically trained to align with human evaluator judgments, achieving a Pearson correlation of 0.897 with human assessments.

4. **GEval with Qwen** (Liu et al., 2023): A flexible evaluation paradigm using chain-of-thought (CoT) prompting with general-purpose instruction-tuned models like Qwen3 (Yang et al., 2025). The model generates detailed reasoning about response quality before assigning a numerical score, enabling GPT-4-level evaluation without specialized training.

Both approaches evaluate responses on multiple criteria using a 1-5 scoring scale, providing both numerical scores and textual feedback explaining the assessment.

Evaluation Methodologies

Multilingual-Prometheus:

- Uses models specifically fine-tuned for multi-criteria evaluation
- Supports multiple model sizes (3B, 7B, 14B parameters)
- Evaluates based on structured rubrics with detailed score descriptions
- Provides both reference-based and reference-free evaluation
- Generates explanatory feedback alongside numerical scores

GEval with Qwen:

- Leverages general-purpose instruction-tuned models for evaluation
- Uses chain-of-thought prompting to elicit detailed reasoning
- Generates evaluation rationale before assigning scores
- More flexible for custom criteria and evaluation frameworks
- Can be adapted to emerging evaluation needs

The choice between approaches depends on evaluation goals: Prometheus offers consistency through specialized training, while GEval provides adaptability through prompting strategies.

Evaluation Rubrics

Both evaluation approaches use a common rubric structure where each criterion is translated into a 1-5 scoring scale with detailed descriptions for each score level. The rubrics combine the high-level evaluation criteria described earlier with concrete scoring guidance for LLM judges.

Rubric Structure Example: Summarization

To illustrate how criteria are operationalized into rubrics, consider the **summarization** task-specific criterion:

```
- Criterion: "Does the response cover key points, stay faithful to the sour
ce in the instruction, and is meaningfully shorter?"
- Score 5: Completely faithful, meaningfully shorter, covers all key points
- Score 4: Mostly faithful, shorter, covers key points well
- Score 3: Generally faithful and shorter but has some issues
- Score 2: Significant issues with faithfulness, coverage, or length
- Score 1: Unfaithful, adds information, misses key points, or too long
```

This pattern applies across all generic criteria (helpfulness, truthfulness, harmlessness, language quality) and task-specific criteria (QA, generation, reasoning, etc.), providing LLM judges with clear guidance for score assignment while maintaining consistency across different task types.

Language-Adaptive Evaluation:

The evaluation framework automatically selects appropriate language quality criteria based on the response language. Dutch responses are evaluated for Dutch language quality, English responses for English language quality, ensuring relevant assessment across both languages.

Evaluation Outputs

LLM judge evaluations produce structured outputs containing:

- Numerical scores for each criterion (1-5 scale)
- Textual feedback explaining the assessment
- Metadata linking scores to specific samples, task categories, and languages

These outputs are aggregated alongside traditional metrics, enabling holistic quality assessment that combines quantitative similarity measures with qualitative judgment of response appropriateness, safety, and task-specific excellence.

3.5.2.4 Configuration and Results

- Which evaluation components to execute (EuroEval, traditional metrics, LLM evaluation)
- Model checkpoint paths and loading parameters
- Test dataset location and sampling limits
- Generation parameters for response production
- Specific settings for each evaluation method (batch sizes, sampling parameters)
- Experiment tracking integration (project, entity, tags)

This configuration-driven approach enables consistent evaluation across different model checkpoints while allowing fine-tuned control over computational resources and evaluation depth.

Results and Logging

The evaluation pipeline produces structured outputs organized by evaluation run:

- **Predictions**: Cached model responses with metadata
- **Traditional metrics**: Aggregated scores at multiple granularities
- **LLM evaluations**: Scores and feedback from evaluation models
- **Combined reports**: Unified view of all evaluation dimensions

All results are automatically logged to Weights & Biases when configured, enabling:

- **Experiment comparison**: Track metrics across model iterations
- **Interactive exploration**: Drill down into specific task categories or failure modes
- **Qualitative analysis**: Review LLM feedback and example predictions
- **Progress visualization**: Monitor improvements across training runs

Analysis and Interpretation

The multi-faceted evaluation provides insights at different levels:

5. **Overall Performance**: High-level comparison with baselines
6. **Task-Specific Analysis**: Understanding which instruction types work well
7. **Language-Specific Patterns**: Dutch vs English performance differences
8. **Quality Dimensions**: Separate tracking of helpfulness, truthfulness, harmlessness, and language quality
9. **Dataset Correlation**: Identifying which training datasets contribute most to specific capabilities

3.5.3 Initial evaluation results

In this section we examine initial results from fine-tuning GPT-NL across different experimental configurations. The figures below present results from the following experiments:

- `GPT-NL 26B (epoch 2): our data` The GPT-NL base model trained through epoch-2 annealing (before epoch 3 data inclusion). Fine-tuned with all available instruction fine-tuning data, including the GPT-NL instruct dataset (see this section for details).

- `GPT-NL 26B (epoch 2): Tulu 3 data` The same GPT-NL epoch-2 model fine-tuned with data from the Tulu 3 initiative. This serves as a data distribution baseline to compare our instruction fine-tuning data against.
- `Olmo 32B: our data` The Olmo 2 model (Team OLMo) fine-tuned with all our instruction fine-tuning data. This enables comparison against a larger, more broadly pre-trained base model.
- `Olmo 32B: Tulu 3 data` Olmo 2 fine-tuned with Tulu 3 data, essentially replicating the Olmo-2 work to verify our evaluation approach.
- `GPT-NL 26B (epoch 2) base model` The GPT-NL base model with no fine-tuning, serving as a performance floor.
- `Olmo 32B base model` The Olmo base model with no fine-tuning, serving as a comparable baseline.
- `GPT-NL 26B (epoch 2) RDS+ filtered data` GPT-NL fine-tuned with a curated subset selected via the RDS+ method.
- `GPT-NL 26B (epoch 2) GEval filtered data` GPT-NL fine-tuned with a curated subset selected via the GEval method.
- `GPT-NL 26B (epoch 2) SPIN v2 data only (gptnl_it_v2)` GPT-NL fine-tuned exclusively with instruction datasets created specifically for GPT-NL.
- `GPT-NL 26B (epoch 2) summarization data only` GPT-NL fine-tuned only with available summarization data.
- `GPT-NL 26B (epoch 2): our data + LoRA adaptor` GPT-NL fine-tuned with all available instruction data using Low Rank Adaptation (LoRA) (Hu et al., 2021) instead of full-parameter training.

3.5.3.1 Internal evaluation (LLM-as-a-judge metrics)

The results below compare all fine-tuning variations using the LLM-as-a-judge evaluation approach with GEval and Qwen 3 as the judge model. These evaluations use test data drawn from the same sources as our fine-tuning datasets and are thus from the same data distribution (notably different from the Tulu 3 distribution, which draws from different sources).

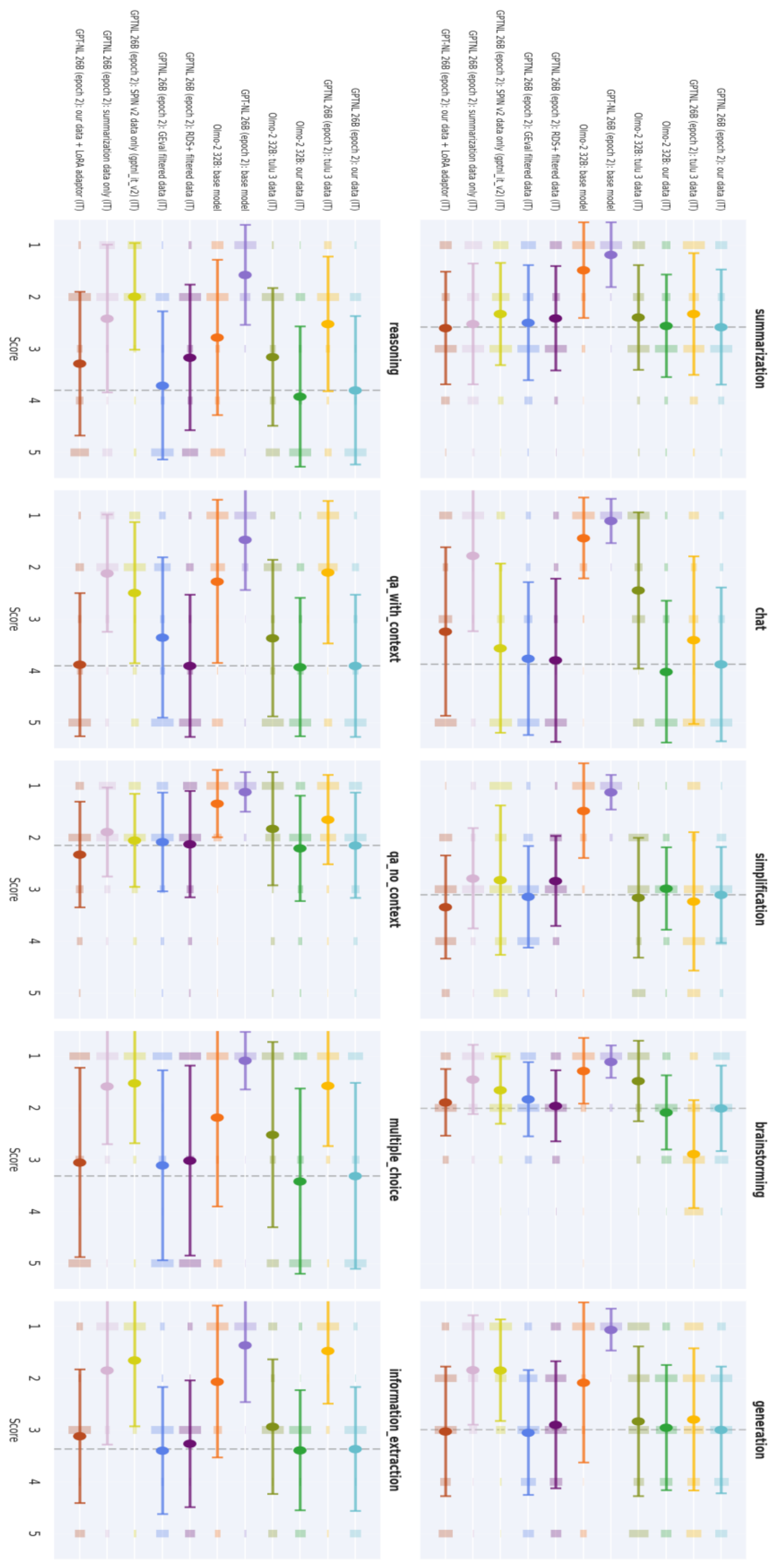
Base Model Performance: The most striking observation is that non-fine-tuned base models (GPT-NL epoch 2 and Olmo-2) substantially underperform all instruction-tuned variants across all task categories. This is expected—base models are optimized for next-token prediction, not instruction following. They lack exposure to this task distribution and thus cannot effectively represent it. Between the two base models, Olmo-2 performs consistently better across all tasks, which reflects its significantly larger pre-training corpus (~6x more tokens than the GPT-NL epoch-2 checkpoint evaluated here). This serves as an important baseline: even with vastly more pre-training data, a base model without instruction fine-tuning remains fundamentally limited for instruction-following tasks.

External Model and Data Comparisons: The experiments swapping either the model (Olmo 2) or data (Tulu 3) with external alternatives (top four rows) show mixed results. Tulu 3 data consistently underperforms our own instruction fine-tuning data across most task categories. This performance gap is primarily attributable to distribution mismatch: Tulu 3 represents these tasks differently than our annotated data does, suggesting that data distribution alignment is critical for fine-tuning success. Notably, using Olmo 2 as the base model provides only marginal improvements over GPT-NL, despite its larger pre-training scale. This suggests that the quality and relevance of instruction fine-tuning data matters more than raw pre-training scale for this task distribution.

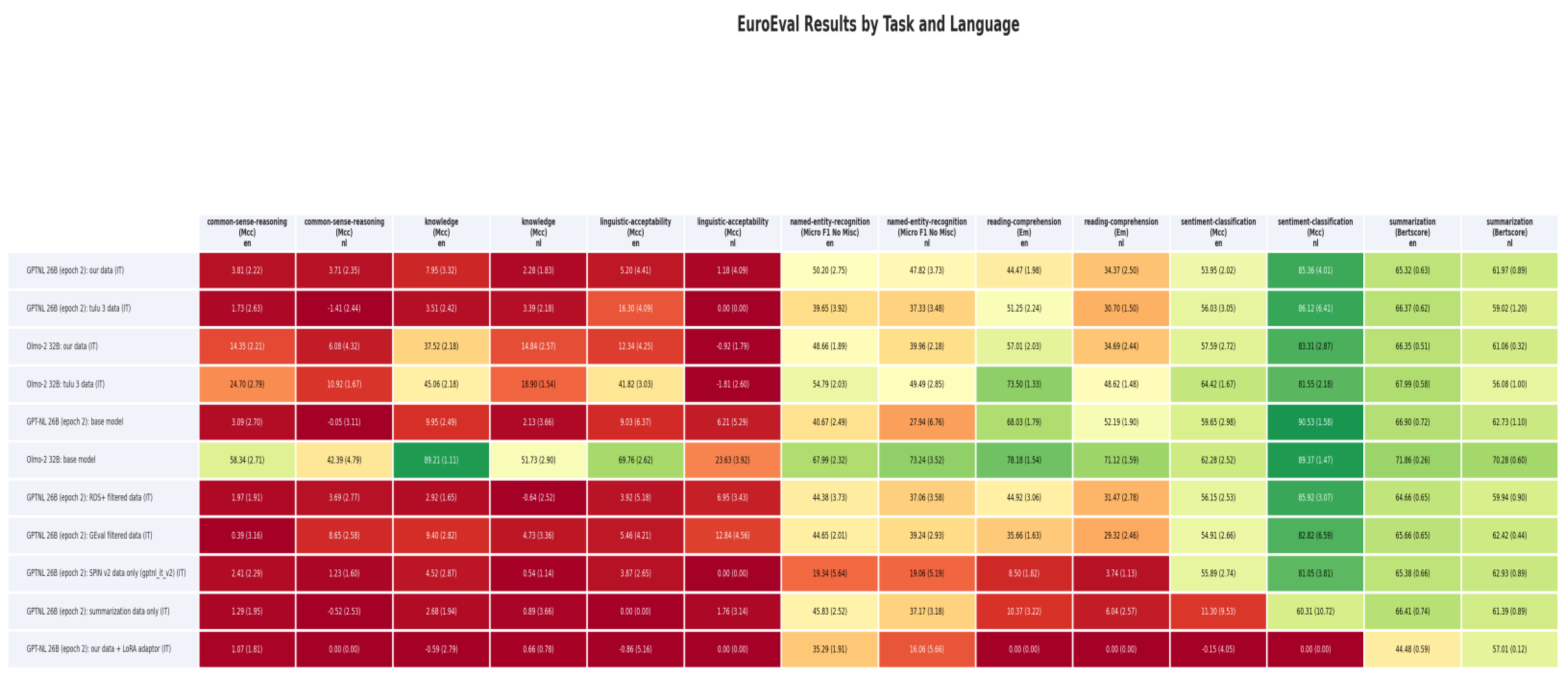
Data Selection and Filtering Experiments: The experiments using filtered data subsets (RDS+ and GEval methods) and domain-specific data subsets (SPIN v2 instruct data only, summarization only) all perform worse than training on the full instruction dataset. This pattern suggests we are operating in a low-data regime where broader coverage is more beneficial than targeted filtering. The consistent underperformance of these variants indicates that our full dataset, despite potential noise, provides valuable diversity that individual task categories or filtering strategies cannot replicate. This finding has important implications: it suggests we should prioritize data quantity and diversity over aggressive quality filtering at this stage.

LoRA Adaptation: The LoRA experiment (fine-tuning with Low Rank Adaptation instead of full parameters) shows no clear advantage over standard full-parameter fine-tuning. While LoRA can be beneficial for parameter efficiency and avoiding catastrophic forgetting, the results here indicate it does not improve task performance on our evaluation set. This may reflect that our dataset size and task complexity benefit from full-parameter optimization, or that the rank constraints of LoRA limit adaptation capability for this diverse task set.

Cross-Task Consistency: Examining performance across task categories (summarization, chat, simplification, brainstorming, generation, reasoning, QA with/without context, multiple choice, information extraction) reveals that performance improvements from instruction fine-tuning are consistent but not uniform. The model shows stronger gains on some task types than others, suggesting specific capabilities are better acquired from our instruction data than others. This variation across tasks provides direction for future data collection—understanding which tasks show smaller improvements can guide targeted data augmentation efforts.


summarization
chat
simplification
brainstorming
generation
reasoning
qa_with_context
qa_no_context
multiple_choice
information_extraction
GPT-NL 26B (epoch 2): our data + LoRA adaptor (IT)
GPTNL 26B (epoch 2): summarization data only (IT)
GPTNL 26B (epoch 2): SPIN v2 data only (gptnl_it_v2) (IT)
GPTNL 26B (epoch 2): GEval filtered data (IT)
GPTNL 26B (epoch 2): RDS+ filtered data (IT)
Olmo-2 32B: base model
GPT-NL 26B (epoch 2): base model
Olmo-2 32B: tulu 3 data (IT)
Olmo-2 32B: our data (IT)
GPTNL 26B (epoch 2): tulu 3 data (IT)
GPTNL 26B (epoch 2): our data (IT)
Score
1
2
3
4
5

3.5.3.2 EuroEval evaluation



EuroEval Results by Task and Language

	common-sense-reasoning (Mcc) en	common-sense-reasoning (Mcc) nl	knowledge (Mcc) en	knowledge (Mcc) nl	linguistic-acceptability (Mcc) en	linguistic-acceptability (Mcc) nl	named-entity-recognition (Micro F1 No Misc) en
GPTNL 26B (epoch 2): our data (IT)	3.81 (2.22)	3.71 (2.35)	7.95 (3.32)	2.28 (1.83)	5.20 (4.41)	1.18 (4.09)	50.20 (2.75)
GPTNL 26B (epoch 2): tulu 3 data (IT)	1.73 (2.63)	-1.41 (2.44)	3.51 (2.42)	3.39 (2.18)	16.30 (4.09)	0.00 (0.00)	39.65 (3.92)
Olmo-2 32B: our data (IT)	14.35 (2.21)	6.08 (4.32)	37.52 (2.18)	14.84 (2.57)	12.34 (4.25)	-0.92 (1.79)	48.66 (1.89)
Olmo-2 32B: tulu 3 data (IT)	24.70 (2.79)	10.92 (1.67)	45.06 (2.18)	18.90 (1.54)	41.82 (3.03)	-1.81 (2.60)	54.79 (2.03)
GPT-NL 26B (epoch 2): base model	3.09 (2.70)	-0.05 (3.11)	9.95 (2.49)	2.13 (3.66)	9.03 (6.37)	6.21 (5.29)	40.67 (2.49)
Olmo-2 32B: base model	58.34 (2.71)	42.39 (4.79)	89.21 (1.11)	51.73 (2.90)	69.76 (2.62)	23.63 (3.92)	67.99 (2.32)
GPTNL 26B (epoch 2): RDS+ filtered data (IT)	1.97 (1.91)	3.69 (2.77)	2.92 (1.65)	-0.64 (2.52)	3.92 (5.18)	6.95 (3.43)	44.38 (3.73)
GPTNL 26B (epoch 2): GEval filtered data (IT)	0.39 (3.16)	8.65 (2.58)	9.40 (2.82)	4.73 (3.36)	5.46 (4.21)	12.84 (4.56)	44.65 (2.01)
GPTNL 26B (epoch 2): SPIN v2 data only (gptnl_it_v2) (IT)	2.41 (2.29)	1.23 (1.60)	4.52 (2.87)	0.54 (1.14)	3.87 (2.65)	0.00 (0.00)	19.34 (5.64)
GPTNL 26B (epoch 2): summarization data only (IT)	1.29 (1.95)	-0.52 (2.53)	2.68 (1.94)	0.89 (3.66)	0.00 (0.00)	1.76 (3.14)	45.83 (2.52)
GPT-NL 26B (epoch 2): our data + LoRA adaptor (IT)	1.07 (1.81)	0.00 (0.00)	-0.59 (2.79)	0.66 (0.78)	-0.86 (5.16)	0.00 (0.00)	35.29 (1.91)

	named-entity-recognition (Micro F1 No Misc) nl	reading-comprehension (Em) en	reading-comprehension (Em) nl	sentiment-classification (Mcc) en	sentiment-classification (Mcc) nl	summarization (Bertscore) en	summarization (Bertscore) nl
GPTNL 26B (epoch 2): our data (IT)	47.82 (3.73)	44.47 (1.98)	34.37 (2.50)	53.95 (2.02)	85.36 (4.01)	65.32 (0.63)	61.97 (0.89)
GPTNL 26B (epoch 2): tulu 3 data (IT)	37.33 (3.48)	51.25 (2.24)	30.70 (1.50)	56.03 (3.05)	86.12 (6.41)	66.37 (0.62)	59.02 (1.20)
Olmo-2 32B: our data (IT)	39.96 (2.18)	57.01 (2.03)	34.69 (2.44)	57.59 (2.72)	83.31 (2.87)	66.35 (0.51)	61.06 (0.32)
Olmo-2 32B: tulu 3 data (IT)	49.49 (2.85)	73.50 (1.33)	48.62 (1.48)	64.42 (1.67)	81.55 (2.18)	67.99 (0.58)	56.08 (1.00)
GPT-NL 26B (epoch 2): base model	27.94 (6.76)	68.03 (1.79)	52.19 (1.90)	59.65 (2.98)	90.53 (1.58)	66.90 (0.72)	62.73 (1.10)
Olmo-2 32B: base model	73.24 (3.52)	78.18 (1.54)	71.12 (1.59)	62.28 (2.52)	89.37 (1.47)	71.86 (0.26)	70.28 (0.60)
GPTNL 26B (epoch 2): RDS+ filtered data (IT)	37.06 (3.58)	44.92 (3.06)	31.47 (2.78)	56.15 (2.53)	85.92 (3.07)	64.66 (0.65)	59.94 (0.90)
GPTNL 26B (epoch 2): GEval filtered data (IT)	39.24 (2.93)	35.66 (1.63)	29.32 (2.46)	54.91 (2.66)	82.82 (6.59)	65.66 (0.65)	62.42 (0.44)
GPTNL 26B (epoch 2): SPIN v2 data only (gptnl_it_v2) (IT)	19.06 (5.19)	8.50 (1.82)	3.74 (1.13)	55.89 (2.74)	81.05 (3.81)	65.38 (0.66)	62.93 (0.89)
GPTNL 26B (epoch 2): summarization data only (IT)	37.17 (3.18)	10.37 (3.22)	6.04 (2.57)	11.30 (9.53)	60.31 (10.72)	66.41 (0.74)	61.39 (0.89)
GPT-NL 26B (epoch 2): our data + LoRA adaptor (IT)	16.06 (5.66)	0.00 (0.00)	0.00 (0.00)	-0.15 (4.05)	0.00 (0.00)	44.48 (0.59)	57.01 (0.12)

Figure 21: EuroEval Evaluation chart

Distribution Shift and Independent Evaluation: EuroEval results paint a markedly different picture from internal evaluation, revealing a critical discrepancy: the independent test distribution substantially challenges all models. Unlike our internal test set—which draws from the same distribution as our fine-tuning data—EuroEval employs standardized, externally-sourced benchmarks across diverse tasks. This distribution shift exposes significant limitations not evident in internal metrics. While our GPT-NL fine-tuned models showed competitive performance on internal tasks, EuroEval reveals widespread underperformance across numerous task categories. This divergence is informative: it indicates our instruction fine-tuning data, while enabling instruction-following capability, may not provide sufficient breadth or quality to generalize to diverse task distributions encountered in practice. The model struggles particularly on certain benchmark tasks (reasoning and knowledge-based tasks), suggesting our training distribution does not adequately cover the patterns present in standardized benchmarks.

Unexpected Experimental Anomaly: A concerning observation emerges when comparing experiments with identical training configurations but different implementation runs. The `Olmo 32B: Tulu 3 data` experiment—which should replicate the original Olmo-2 training (both using Olmo as base model and Tulu 3 data) – shows notably different results than expected. This discrepancy suggests training parameters, random seeds, or infrastructure differences between our implementation and the reference may be affecting reproducibility. This anomaly highlights that either: (1) our experimental setup has undocumented variations affecting training outcomes, or (2) the reference conditions were not precisely replicated. Resolving this is critical for ensuring experimental validity and understanding which design choices actually drive performance improvements.

Metric Disagreement and Evaluation Complexity: A fundamental challenge emerges when comparing internal LLM-as-a-judge metrics with EuroEval's standardized benchmarks: they do not tell the same story. Models that rank highly on internal evaluation often show weaker EuroEval performance, and vice versa. This metric disagreement reflects different evaluation philosophies: our internal LLM judges assess task-specific quality with detailed rubrics aligned to our instruction fine-tuning objectives, while EuroEval employs standardized benchmarks designed for broad model comparison. This divergence suggests that high performance on internally aligned tasks does not guarantee generalization to external benchmarks. Moving forward, we cannot optimize against a single metric without risking misaligned improvements. This necessitates: (1) identifying which metrics best correlate with our actual deployment goals, (2) understanding what EuroEval benchmarks reveal about genuine model limitations, and (3) deciding whether to prioritize internal task-specific excellence or external benchmark generalization.

Implications for Future Work: These findings indicate substantial refinement is needed before declaring results conclusive. First, the experimental parameter anomaly must be investigated and resolved to ensure reproducibility. Second, we need to systematically understand which EuroEval tasks show the largest gaps and why, whether due to distribution mismatch, insufficient training data, or model capacity limitations. Third, we should establish core evaluation metrics that balance internal task performance with external benchmark robustness, avoiding optimizing for one dimension at the expense of another. The current results suggest we are still in an exploratory phase where different experimental choices lead to different rank orderings across evaluation dimensions, indicating the need for more targeted experimentation to achieve stable, reproducible improvements across multiple evaluation perspectives.

3.6 Code and Data Organization

The instruction fine-tuning infrastructure is organized to support efficient experimentation with multiple base models, dataset variants, and training configurations. The codebase is maintained in the Instruction Fine-Tuning Repository, while all training data and model artifacts are stored on the Snellius HPC cluster.

The organization reflects the iterative nature of fine-tuning research: individual datasets are prepared once from their source formats, then combined into different training mixtures for experimental runs. Configurations are version-controlled separately from code, and training outputs are systematically organized to enable comparison across runs. This structure supports rapid experimentation with different data combinations while maintaining reproducibility.

The Instruction Fine-Tuning Repository contains the complete pipeline for GPT-NL instruction fine-tuning, built on HuggingFace TRL and scaled to multi-node training with DeepSpeed. The repository implements data preparation, distributed training, and comprehensive evaluation capabilities.

The repository serves as the central location for all instruction fine-tuning workflows. It provides:

- **Data processing pipeline**: Transform heterogeneous instruction datasets into unified training format
- **Training infrastructure**: Distributed fine-tuning with DeepSpeed ZeRO on multi-node clusters
- **Evaluation framework**: Both traditional metrics and LLM-based quality assessment
- **Experiment management**: YAML-based configuration and Weights & Biases tracking

The implementation prioritizes modularity and configurability, allowing researchers to easily experiment with different data mixtures, training recipes, and evaluation strategies without modifying core code.

3.6.1 Data Folder Structure

The data organization follows these principles to support reproducibility and efficient experimentation:

- **IT mixture datasets** (`it_mixtures/`): Post-processed, combined dataset variants ready for fine-tuning. Each mixture represents a specific data selection strategy (e.g., `all_data`, `rds_plus_178k`, `conversational`) and can be directly loaded for training.

- **Raw IT datasets by split** (`it_datasets/train/`, `val/`, `test/`): Individual dataset splits organized by source before combining into mixtures. This separation allows flexible mixture creation without re-downloading or re-processing source datasets.

- **Pretrained models** (`pretrained-models/`): Base model checkpoints (GPT-NL and external models like OLMo) used as starting points for fine-tuning. Organized by model name and training stage (e.g., `epoch_2_annealed_step98863`).

- **Fine-tuned checkpoints** (`it-checkpoints/`): Model checkpoints saved during and after training. Naming convention: `{BASE_MODEL}-{DATASET}-{BATCH_SIZE}-gas-{GAS}-nodes-{NODES}-{ZERO_STAGE}-{SLURM_JOB_ID}` (e.g., `GPTNL-26B-`

all_data-4-gas-1-nodes-8-zero3-15593638). Each checkpoint folder contains model weights, optimizer states, and training metadata.

- **Evaluation results** (eval_results/): Outputs and metrics organized by SLURM job ID with descriptive folder names: {JOB_ID} BASE=[...] IT_DATA=[...] EVAL=[...]. This structure enables easy identification and comparison of evaluation runs.
- **Chat templates** (chat_templates/): Jinja2 templates for different model families that define how to format multi-turn conversations. Used during both training and inference to ensure consistent formatting.

3.6.2 Source Code Structure

The codebase follows a modular structure that cleanly separates concerns: configuration files define what to run, job scripts define where and how to run it, and source code implements the logic. This separation enables easy experimentation and deployment across different computing environments:

```
instruction-finetuning/
├─ config/                         # YAML configuration files for data proces
sing, training and evaluating
│   ├─ data_processing/
│   ├─ train/
│   ├─ evaluate/
├─ jobs/                           # SLURM job scripts for Snellius
│   ├─ train/
│   │   ├─ distributed_sft.sh      # Main distributed training launcher
│   │   ├─ distributed_grid_launcher.sh  # Grid search launcher
│   │   ├─ torchrun_launcher.sh    # PyTorch distributed launcher
│   │   └─ convert_zero_to_fp32.sh # Convert DeepSpeed checkpoints to FP32
│   ├─ evaluate/                   # To start evaluation runs
│   └─ misc/                       # Miscelaneous, for example to set up the
environment or start RDS+ data selection process
├─ src/                            # Python source code
│   ├─ data_processing/            # Data preparation pipeline
│   ├─ train/                      # Training implementation
│   ├─ it_evaluate/                # Evaluation framework
│   └─ utils/                      # Shared utilities
├─ pyproject.toml                  # Python dependencies (uv package manager)
├─ README.md                       # Repository documentation
└─ .env_example                    # Environment variable template
```

Configuration System: YAML-based configuration using OmegaConf for environment variable substitution and hierarchical configs. Training, evaluation, and data processing are separately configured.

Data Processing Pipeline: Modular pipeline that downloads, standardizes, unrolls (converts to prompt-completion format), applies chat templates, and filters datasets. See Data Preparation for details.

Training Implementation: Built on HuggingFace TRL's SFTTrainer with DeepSpeed ZeRO for distributed training. Supports multi-node training on Snellius with automatic checkpoint management. See Training for implementation details.

Evaluation Framework: Dual-track evaluation with traditional metrics (BLEU, ROUGE, BERTScore) and LLM-based evaluation (Prometheus, G-Eval). Integrates with Weights & Biases for tracking. See Evaluation for methodology.

Utilities: Shared argument parsing (`args.py`), model loading with flash attention support (`model_utils.py`), and distributed training helpers (`distributed_utils.py`).

3.6.3 Dependencies

Dependencies are managed via `pyproject.toml` using the uv package manager for fast, reliable environment setup:

- **Core**: `transformers==4.57.1`, `trl>=0.17.0`, `torch>=2.0.0`, `datasets==3.6.0`
- **Training**: `deepspeed>=0.12.0`, `accelerate>=0.24.0`
- **Evaluation**: `prometheus-eval>=0.1.20`, `euroeval==15.16.0`, `bert-score>=0.3.13`, `vllm==0.10.0`
- **Tracking**: `wandb>=0.15.0`
- **Configuration**: `omegaconf>=2.3.0`, `pyyaml>=6.0.0`

3.6.4 Practical Information

Installation

For Snellius deployment, use the automated installation script:

```
cd instruction-finetuning
./jobs/misc/install_snellius.sh
```

This script sets up a virtual environment, installs all dependencies via `uv`, and configures the environment for distributed training.

Configuring and Starting Jobs

Training and evaluation jobs are configured through YAML files in the `config/` directory, which specify model paths, dataset locations, hyperparameters, and resource requirements. Job scripts in `jobs/train/` and `jobs/evaluate/` contain SLURM directives for compute resources (nodes, GPUs, time limits) and load the appropriate configuration files. Jobs are submitted to the Snellius queue using `sbatch jobs/train/<script>.sh`, which launches distributed training across the requested compute nodes.

Logs and Checkpoints

During training, checkpoints and logs are automatically saved to the shared project space (`/projects/0/prjs0986/wp14/instruction-finetuning/it-checkpoints/`) under descriptive folder names following the naming convention: `{BASE_MODEL}-{DATASET}-{BATCH_SIZE}-gas-{GAS}-nodes-{NODES}-{ZERO_STAGE}-{SLURM_JOB_ID}`.

Each training run directory contains periodic checkpoints saved at configured intervals (e.g., `checkpoint-100/`, `checkpoint-200/`) with model weights, optimizer states, and training metadata. If Weights & Biases tracking is enabled, run data is stored in a `wandb/` subdirectory.

Evaluation outputs are organized under `eval_results/` with folders named by SLURM job ID and descriptive metadata: `{JOB_ID} BASE=[model] IT_DATA=[dataset]`

`EVAL=[test_set]`. Each folder contains model predictions, computed metrics, and references to the source checkpoint.

4 Model Deployment

We deploy GPT-NL behind an OpenAI-compatible HTTP API provided by vLLM, with two client applications on top:

Gradio “Ops UI”: a lightweight control panel to start and tune runtime knobs (sampling, limits, concurrency) and run quick smoke-tests/benchmarks.

Open WebUI Chat: a full-featured chat workspace for end users (multi-user, permissions, conversation UX), connected to the same OpenAI-compatible endpoint.

vLLM is optimized for high-throughput, GPU-efficient serving via:

- PagedAttention (KV-cache memory efficiency) and continuous batching of incoming requests (better GPU utilization under concurrent load).
- Production-serving features like streaming outputs, prefix caching, and multi-LoRA support.
- A broad set of performance knobs including quantization options (e.g., GPTQ, AWQ, INT4/INT8/FP8) plus features like speculative decoding and chunked prefill (model- and workload-dependent).
- A built-in OpenAI-compatible API server, enabling drop-in compatibility with OpenAI SDK-based clients.
- Seamless integration with HuggingFace checkpointed models (like GPT-NL)
- Broad hardware compatibility like NVIDIA GPUs, AMD CPUs and GPUs, Intel CPUs and GPUs, PowerPC CPUs, Arm CPUs, and TPU.

Evaluation deployment

To evaluate both the pre-trained and fine-tuned GPT-NL model, we needed a fast-serving module that could work as “plug-and-play” component on various tasks, e.g. offline Euroeval.

In practice, we have created an Apptainer image with a self-contained vLLM instance. The creation recipe is available here, and the image is located on `/projects/0/prjs0986/wp14/containers/vllm_25.09.sif`.

Deploying this image from cli can be performed with the below command. For the specific variables please look here.

```
apptainer exec --nv -B $PROJECT_SPACE -B $DOWNLOAD_DIR $CONTAINER_PATH \
    vllm serve $MODEL_CHECKPOINT \
    --tensor-parallel-size $GPUS --download-dir $DOWNLOAD_DIR \
    --uvicorn-log-level warning --chat-template $CHAT_TEMPLATE
```

Demo deployment

For Demo purposes, we developed a 2-stage deployment setup.

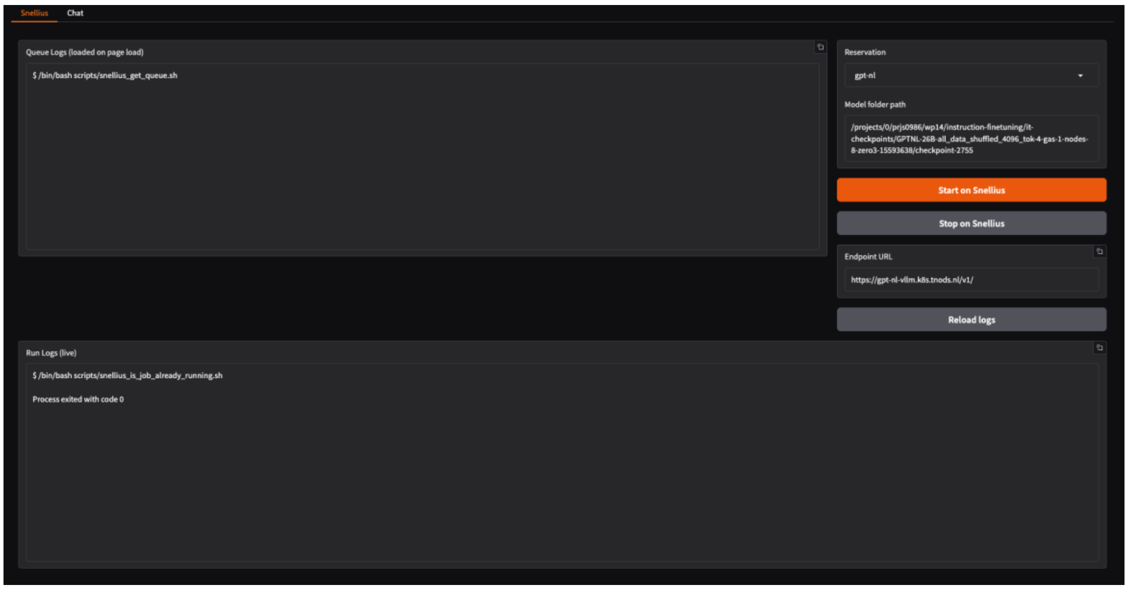
1. Demo starter web application
2. Demo Chat application

In more detail (available at: https://gpt-nl-demo-starter.k8s.tnods.nl/):

This is a Gradio web application that allows the user to:

- Check whether there is a running vLLM instance
- Select the reservation on snellius (“gpt-nl” or “”)
- Select which model version / checkpoint to run from Snellius
- Start / Stop the GPT-NL model serving on Snellius
- Exposes the endpoint within TNO network
- Offers a Chatbox window for using the model
- Provides a Tab with user-adaptable parameters (temperature, top-p, min-p, repetition penalty)
- Provides a Tab with monitoring metrics (prompt throughput, generation throughput (tokens/sec), total tokens)

Codebase and deployment instructions available here.



Queue Logs (loaded on page load)

```
$ /bin/bash scripts/snellius_get_queue.sh
```

Reservation: gpt-nl

Model folder path: /projects/0/prjs0986/wp14/instruction-finetuning/it-checkpoints/GPTNL-26B-all_data_shuffled_4096_tok-4-gas-1-nodes-8-zero3-15593638/checkpoint-2755

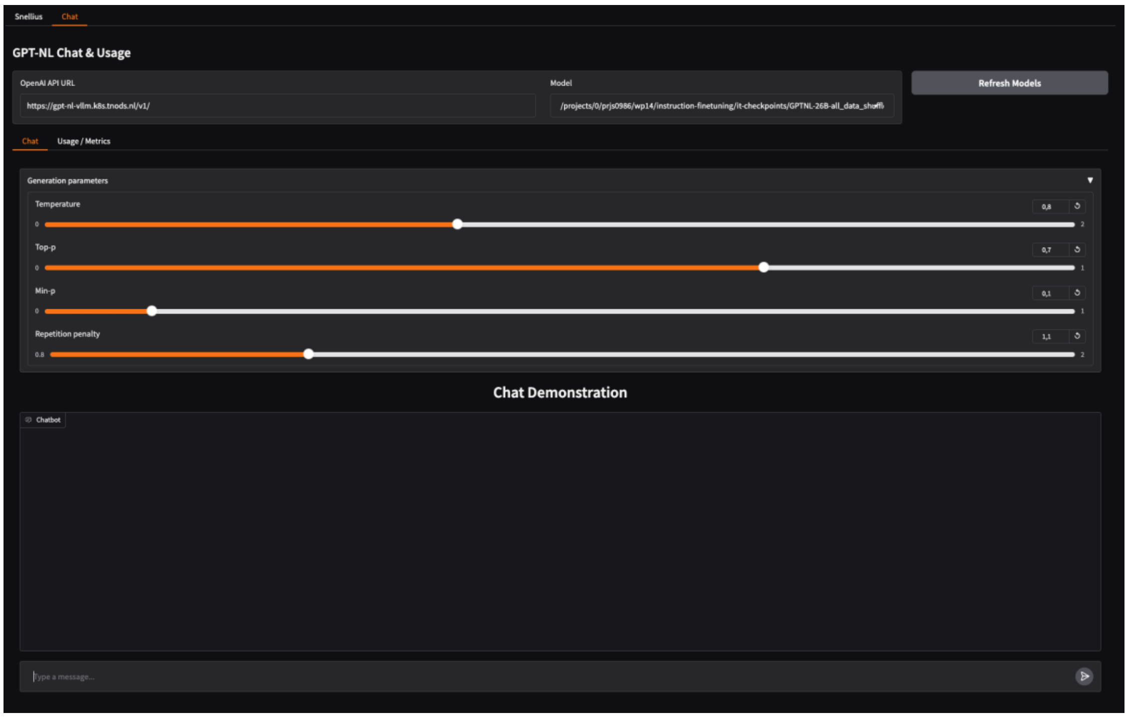
Endpoint URL: https://gpt-nl-vllm.k8s.tnods.nl/v1/

Run Logs (live)

```
$ /bin/bash scripts/snellius_is_job_already_running.sh

Process exited with code 0
```

Figure 22: GPT-NL Demo GUI (screenshot)



GPT-NL Chat & Usage

OpenAI API URL: https://gpt-nl-vllm.k8s.tnods.nl/v1/

Model: /projects/0/prjs0986/wp14/instruction-finetuning/it-checkpoints/GPTNL-26B-all_data_shuffl

Generation parameters

Temperature	0,8	0	2
Top-p	0,7	0	1
Min-p	0,1	0	1
Repetition penalty	1,1	0.8	2

Chat Demonstration

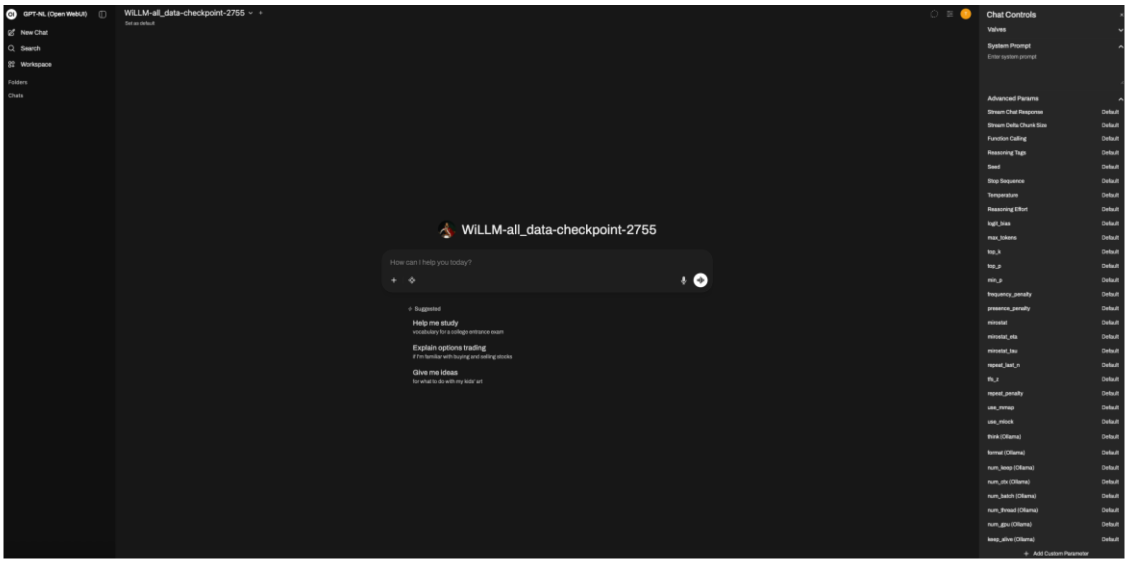
Figure 23: GPT-NL Demo GUI (Screenshot 2)

Demo Chat application

Available at: http://gpt-nl-chat.tnods.nl/

Open WebUI is a user-friendly platform that offers offline operations and works with various LLM runners like Ollama and OpenAI-compatible APIs.

In our case, we are using the endpoint exposed from the Demo starter and enhance the user-experience with a “ChatGPT” style interaction.



WiLLM-all_data-checkpoint-2755

Stream Chat Response	Default
Stream Delta Chunk Size	Default
Function Calling	Default
Reasoning Tags	Default
Seed	Default
Stop Sequence	Default
Temperature	Default
Reasoning Effort	Default
logit_bias	Default
max_tokens	Default
top_k	Default
top_p	Default
min_p	Default
frequency_penalty	Default
presence_penalty	Default
mirostat	Default
mirostat_eta	Default
mirostat_tau	Default
repeat_last_n	Default
tfs_z	Default
repeat_penalty	Default
use_mmap	Default
use_mlock	Default
think (Ollama)	Default
format (Ollama)	Default
num_keep (Ollama)	Default
num_ctx (Ollama)	Default
num_batch (Ollama)	Default
num_thread (Ollama)	Default
num_gpu (Ollama)	Default
keep_alive (Ollama)	Default

Figure 24: GPT-NL Demo chat application

The Demo starter web application does not interact with Snellius directly; instead, it calls a small set of scripts that hide all cluster-specific logic. The UI uses `snellius_get_queue.sh` (and a companion “is job running” script) to show SLURM queue status and detect existing vLLM jobs, and `snellius_start.sh` / `snellius_stop.sh` to start or stop the remote vLLM server. These scripts delegate the actual job submission, lifecycle management, and SSH tunneling to `remote_vllm_manager.py`, which in turn submits a SLURM job using `spawn_vllm_slurm_snellius.job` to launch the Apptainer-based vLLM server with the selected model checkpoint. The Gradio backend (`vllm_serve_server_base.py`) simply streams logs from these scripts into the UI and enables/disables the controls based on their output. For full implementation details, see the code and comments in the demo starter repository.

5 Appendices

This chapter provides a collection of technical reference materials, including hardware specifications, detailed software stack evaluations, assessment results, and formal data and model format definitions. These resources support the system architecture activities but are too detailed to include in the main body of the document.

Each section in this chapter consolidates essential technical information on topics related to the system architecture work. They serve as reference points for the main sections of the report, offering detailed substantiation for their content.

The following appendices are included:

- GPT-NL data curation and training at SURF's HPC Snellius
- Scaling the training in the HPC
- Frameworks for GPT-NL Fine-Tuning

5.1 GPT-NL data curation and training at SURF's HPC *Snellius*

Snellius serves as the **national supercomputer** managed by SURF for the Dutch high-performance computing (HPC) community. Designed to support both academic and industrial research, Snellius delivers cutting-edge, heterogeneous computing capabilities—from CPU-only nodes leveraging AMD's Rome and Genoa architectures to GPU-accelerated configurations with NVIDIA A100 and H100 devices. This system plays a pivotal role in enabling large-scale, data-intensive simulations, machine learning applications, and scientific computing across the Netherlands. With robust SLURM-based job scheduling, flexible partitioning, and precise accounting in System Billing Units (SBUs), Snellius empowers users to maximize computational throughput while maintaining transparency and efficiency—making it a cornerstone of Dutch HPC infrastructure.

These are the key Snellius Partitions used in GPT-NL project.

5.1.1 Standard nodes

rome (alias thin)

- **Node type**: Thin compute nodes (tcn)
- **CPU**: AMD Rome, 128 cores/node
- **Memory**: 224 GiB usable RAM/node
- **Allocation granularity**: 1/8 node ≈ 16 cores + 28 GiB RAM

genoa

- **Node type**: Thin compute nodes (tcn)
- **CPU**: AMD Genoa, 192 cores/node
- **Memory**: 336 GiB usable RAM/node
- **Allocation granularity**: 1/8 node ≈ 24 cores + 42 GiB RAM

- **Main usage in GPT-NL**: Tests, development, and data curation

5.1.2 GPU-Accelerated Partitions

gpu_A100

- **Node type**: GPU compute nodes (gcn)
- **CPU**: Intel Xeon Platinum 8360Y, 72 cores/node
- **Memory**: 480 GiB RAM/node
- **GPU**: 4 × NVIDIA A100 (40 GB each)
- **Allocation granularity**: 1/4 node ≈ 18 cores + 1 GPU + 120 GiB RAM
- **Main usage in GPT-NL**: Tests, development, model training, and data curation

gpu_H100

- **Node type**: GPU compute nodes (gcn)
- **CPU**: AMD EPYC 9334, 64 cores/node
- **Memory**: 720 GiB RAM/node
- **GPU**: 4 × NVIDIA H100 (94 GiB each)
- **Allocation granularity**: 1/4 node ≈ 16 cores + 1 GPU + 180 GiB RAM
- **Main usage in GPT-NL**: Tests, development, model training, and data curation. Most of the pre-training and fine-tuning phases used the gpu_H100 partition with exclusive reservations of up to 22 nodes for longer training batches.

These configurations enable flexible, high-performance computing suitable for a wide range of scientific and engineering applications, reflecting Snellius's role as a versatile and advanced national HPC asset.

5.2 Scaling the pre-training at Snellius

To train the GPT-NL model, we need to scale the 26B parameter model across the multi-node GPU cluster on Snellius.

In the ideal set-up, we scale to the maximum number of nodes available. We also require a flexible set-up since individual nodes can become temporarily unavailable due to hardware failure or maintenance.

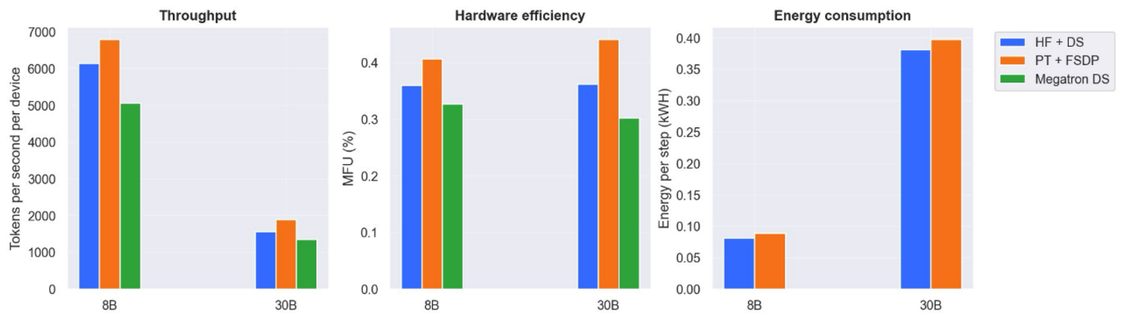
The main pre-training runs were performed on 22 NVIDIA H100 (4 x 94GiB HBM2e) nodes interconnected via Infiniband HDR100 (100Gbps).

5.2.1 FSDP

To do so, we use Fully Sharded Data Parallelism (FSDP) implemented in PyTorch FSDP2 (Zhao et al., 2023). FSDP works by distributing (sharding) the model parameters, optimizer states, and gradients across multiple workers, so each worker holds only a portion of the model rather than a full replica as in DDP (Distributed Data Parallel). This reduces the memory footprint on each GPU, enabling the training of larger models or batch sizes, while internal optimizations like overlapping communication with computation help mitigate the increased communication overhead that is added when sharding the model over multiple nodes. Activation checkpointing is used to reduce memory consumption, allowing for larger batch sizes.

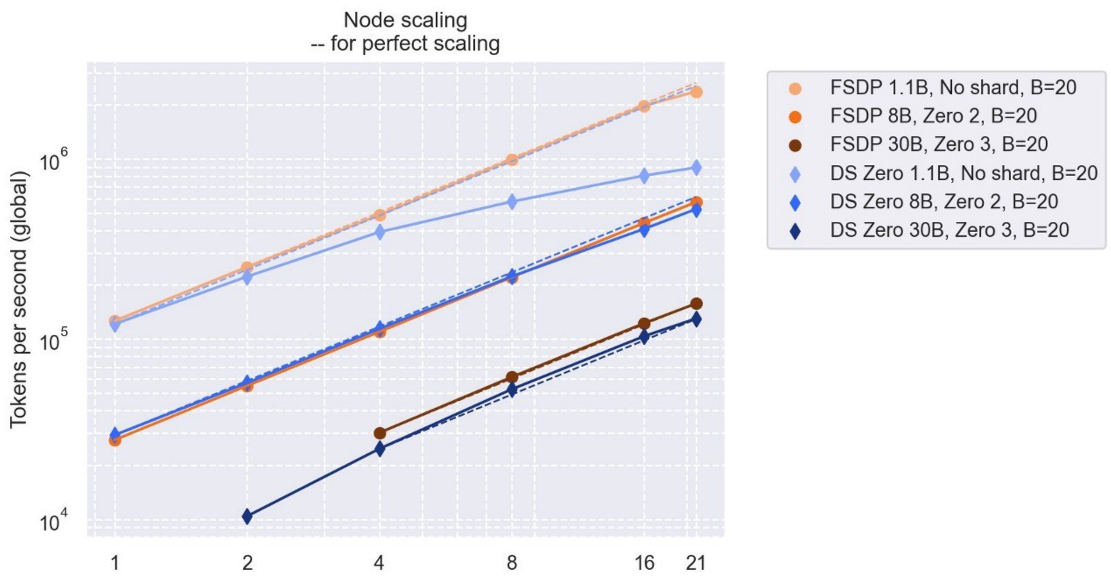
Experiments

To investigate the different approaches outlined in the documentation on sharding, we ran experiments to test the scaling of the models using the different sharding strategies and frameworks. The following figure shows the performance results for training 8B and 30B parameter models (with a similar architecture to the final GPT-NL model) on 20 nodes:



Based on these experiments, FSDP showed the best efficiency and throughput (i.e. how many tokens can be processed per second). These results were consistent over the increasing model sizes.

Additional experiments were performed to investigate the scaling efficiency when increasing the number of nodes from 1 to 22. The models were trained with increasing levels of sharding: no sharding (data parallel), Zero2 (for FSDP, this is the `SHARD_GRAD_OP` setting) and Zero3 (FSDP:`FULL_SHARD`).



The dotted line in the graph shows the ideal scaling, when adding more nodes would add linear improvement to the total throughput. In practice, some performance degradation is expected because of communication overhead. In particular, with increasing sharding stages this communication is expected to increase. For each of the sharding stages, the FSDP approach proved more efficient and showed better throughput. The decrease in throughput also proved minimal, due to fast interconnect between the nodes.

Concluding from these experiments, for the final model training we opted for full sharding with FSDP.

Parameters

Below is an overview of the key configuration parameters:

Parameter	Value / Expression	Description
Parallelism strategy	`FSDP`	Fully Sharded Data Parallel. Model weights, gradients, and optimizer state are sharded across GPUs to minimize memory footprint.
Cluster setup	22 nodes x 4 H100 GPUs per node	Distributes training load across 88 GPUs in total, enabling distributed training at scale.
Parameter precision	`DType.bfloat16`	Parameters are stored in **bfloat16**, helping reduce memory usage while maintaining numeric stability.
Activation checkpointing	`TransformerActivationCheckpointingMode.full`	Applies full checkpointing of activations to save memory during the forward/backward passes by recomputing forward activations in the backward pass.
Flash attention	`True`	Enables FlashAttention v2 kernels for efficient attention computation.
Per-device batch size	`12`	Each GPU processes 12 sequences per step, balancing throughput, and memory capacity.
Gradient accumulation steps	`3`	Accumulates gradients over 3 micro-batches before performing an optimizer step,

		effectively increasing the global batch size without exceeding memory limits.

5.2.2 Checkpointing

To ensure a flexible restarting schedule, we ensure that (temporary) checkpoints are stored every 220 steps, allowing for continued training with a potentially (temporary) change in number of nodes.

5.3 Frameworks for GPT-NL Fine-Tuning

This appendix presents a concise study conducted within the GPT-NL project to identify and select one or a small set of frameworks suitable for the fine-tuning stage. For the GPT-NL fine-tuning phase, the primary objective was to perform a full fine-tune using Supervised Fine-Tuning (SFT), as this approach is widely recognized for its effectiveness in adapting large language models to domain-specific tasks while maintaining stability.

The selected framework must fully support this approach. Among the feasible options, the final choice was based on performance considerations. The evaluation criteria included scalability, ease of integration, community support, and computational efficiency. Other alignment techniques, such as Direct Preference Optimization (DPO) and Generalized Reinsertion Policy Optimization (GRPO), were outside the scope of this study. However, these methods were kept in mind during the framework evaluation to ensure future compatibility. The study compared several candidate frameworks commonly used for fine-tuning large language models:

HuggingFace TRL – A widely adopted library offering strong support for SFT and reinforcement learning-based alignment. Open-Instruct (OLMo) – Focused on open-source instruction fine-tuning with robust tooling for research workflows. ColossalAI – Designed for large-scale distributed training with efficient memory optimization. DeepSpeedChat – Provides advanced optimizations for chat-based fine-tuning and large-scale deployments. Axolotl – A lightweight solution tailored for LoRA and parameter-efficient fine-tuning. Megatron-LM – Optimized for massive model training with tensor and pipeline parallelism. TorchTune – A PyTorch-native library emphasizing simplicity and modularity for fine-tuning tasks. Unsloth – Specializes in fast and resource-efficient fine-tuning, particularly for smaller hardware setups.

5.3.1 Frameworks Comparison

These are the frameworks that we found:

Framework	Description / Focus	Multi-node & Sharding	Multi-GPU	PEFT (LoRA)	Full Finetune	SFT	DPO	GRPO	Notable Models Trained
Huggingface TRL	Fine-tuning and alignment (SFT, DPO, PPO). Built on HF + Accelerate + DeepSpeed. Good balance between boilerplate and modularity	✓ ZeRO FSDP	✓	✓	✓	✓	✓	✓ (ORPO)	OpenAssistant
Open-Instruct / olmo-core	AllenAI's framework for instruction fine-tuning. Focused on RLHF at scale.	✓ FSDP	✓	✓	✓	✓	✓	✓ (RLVR)	OLMo-2 32B Instruct
ColossalAI	High-performance training with 3D parallelism. Pretraining and finetuning at scale.	✓ FSDP	✓	✓	✓	✓	✓	✓ ORPO, KTO	OpenChat 3.5, Colossal-LLaMA

DeepSpeed-Chat	High-efficiency distributed training. Focus on full finetuning & RLHF (SFT, PPO, RM). Excellent performance.	✅ ZeRO-1/2/3, multi-node	✅	Limited	✅	✅	✅ (via custom setup)	❌ (no official support)	Salamandra
Axolotl	Config-based wrapper for HF. Easy SFT/LoRA/full finetune.	✅ ZeRO, FSDP	✅	✅	✅	✅	✅	✅	EuroLLM-9B, OpenHermes
Megatron-LM	Scalable pretraining & full finetuning. Performance-optimized. No alignment-specific tools.	✅ FSDP, DP, PP, SP	✅	❌ (in NeMO yes)	✅	✅	❌	❌	Megatron-Turing NLG 530B, GPT-NeoX
TorchTune	Modular PyTorch-native tuning. Research-friendly. Early but growing.	❌ (no DPO)	✅	✅	✅	✅	✅	✅	Llama?
Unsloth	High-performance, optimized fine-tuning & RLHF; low VRAM, fast speed-ups	✅ Multi-node (Paid enterprise)	✅	✅ LoRA & QLoRA	✅	✅	✅	✅	

After analysing multiple frameworks for instruction fine-tuning, including HuggingFace TRL, ColossalAI, DeepSpeedChat, Open-Instruct, and others, we chose **HuggingFace TRL** as our **primary fine-tuning framework**, augmented with:

- **DeepSpeed ZeRO** for multi-GPU/multi-node scaling (Stages 1–3).
- **Dataset streaming** (for memory efficiency on large parquet datasets).
- **Synchronization primitives** (avoid races in model/dataset caching across many ranks).
- **Custom callbacks** for **throughput** and **sample generation** into monitoring tools.

See further discussion in the Fine-tuning training Section.

5.3.2 RLHF approaches guideline

Method	Best For	Pros	Cons
PPO (Proximal Policy Optimization)	Full RLHF with token-level control and dynamic environment interaction	✅ Fine-grained learning ✅ Proven for large models like ChatGPT	❌ Expensive (needs reward + value model) ❌ Sensitive to reward model tuning
DPO (Direct Preference Optimization)	Simple preference alignment tasks, especially single-turn dialogue	✅ Stable, efficient ✅ No RL or reward model needed	❌ Limited to pairwise preferences ❌ Less expressive for long-horizon tasks
GRPO (Generalized Reinsertion	Reasoning-heavy, long-horizon tasks where PPO is unstable	✅ No value model required	❌ New and less widely adopted

Policy Optimization)		✅ More stable and sample-efficient than PPO	
RLVR (RL with Verifiable Rewards)	Tasks with objective success signals (math, code correctness)	✅ Uses true/automated rewards ✅ Less dependent on human feedback	❌ Only suitable for tasks with measurable outputs
ORPO (Offline RL with Policy Optimization)	RLHF-style training with static datasets (no rollouts)	✅ Efficient offline tuning ✅ Leverages existing reward models	❌ No exploration ❌ Limited to seen data
KTO (KL-Tuned Optimization)	Reward-guided fine-tuning without full RL setup	✅ Lightweight and easy ✅ Ideal for hybrid supervised + reward training	❌ Less powerful than PPO for complex behavior

5.3.3 References

- AllenAI released SFT, DPO and Instruct/GRPO versions of their 32B model
- Trained on 5 8xH100 nodes source
- Fine-Tuning LLMs with GRPO on AMD MI300X: Scalable RLHF with Hugging Face TRL and ROCm - link
- DeepSpeedChat SFT, DPO, RM finetune, RLHF - link
- ColossalAI example scripts for PPO, DPO, GRPO, etc. - link
- OLMo-32B RLVR - link