# CSE 5525 (Spring 2026) Default Final Project

CSE 5525

## 1 Overview and Motivation

Modern large language models based chat assistants typically undergo a multi-stage *post-training* pipeline:

1. Supervised fine-tuning (SFT) on instruction–response data.
2. Alignment using human preferences or reward signals (e.g., DPO, RLHF, RLVR).
3. Careful evaluation to understand helpfulness, safety, and robustness tradeoffs.

In this project, you will reproduce a simplified version of this pipeline using a ∼1B parameter open LLM provided on Tinker. Your goal is to:

- build a correct and reproducible training pipeline,
- understand how SFT and RL-style alignment change model behavior,
- analyze where alignment helps, where it hurts, and why.

This project mirrors published workflows used to train models such as Qwen, Llama, OLMo, etc. but at a scale suitable for a course. While commercial AI assistants like ChatGPT, Claude, and Gemini do not publish their training recipes, they likely follow a similar pipeline as well.

**Note:** It is not intended for the default final project to require less effort or work than the custom final project. The default project simply excludes the difficulty of devising your own project idea and evaluation methods, allowing students to commit an equivalent amount of effort to the provided problem.

## 2 Teams, Compute Budget, and Practical Constraints

**Teams.** You are encouraged to work in teams of 2–3 students. Larger teams are expected to run more experiments and perform deeper analysis. Teams of 1 or >3 require permission from the instructor.

**Training infrastructure (Tinker).** All training in this project will be performed using **Tinker**, a training API developed by Thinking Machine Labs. Tinker handles the heavy computation for forward and backward passes on remote GPU infrastructure, allowing you to run experiments from your local machine (e.g., your laptop) without direct access to GPUs.

Each enrolled student will receive:

- a personal **Tinker API key**, distributed by email (if you have not received this yet, please email the instructor/TA), and
- a fixed amount ($250) of **Tinker training credits** for use during this project.

The credit cost of each training/inference is charged per (million) tokens, which are different for each model, and training method (such as, SFT vs RL). Please check training costs here: `https://thinkingmachines.ai/tinker/` (under FAQ). Since you only have $250 in credits, you are expected to design experiments that are **compute-efficient** and to justify all major training runs in your report. These credits should be more than enough to obtain the expected results.

**Monitoring usage.** All training on Tinker is done using Low rank adapters for efficiency reasons (we will discuss them in class soon). You are responsible for:

- tracking your Tinker credit usage,
- stopping runs that are clearly misconfigured,
- budgeting credits across SFT, alignment, and evaluation stages.

Exceeding the credit limit or exhausting credits early will not result in additional allocations, so plan experiments carefully.

**Model.** All teams must start from the same base model provided on Tinker (`Llama-3.2-1B`) to receive the base grade. You may choose to use any other available model for exploratory sections.

**Data integrity constraint.** You **may not**:

- train on any held-out test sets,
- tune hyperparameters using test labels,
- manually edit outputs after generation.

Any form of test leakage will result in a failing grade.

## 3 Starter Code and Provided Infrastructure

We will provide a starter repository that implements most low-level engineering details. You are expected to extend this code base. We will update this document with more details in the coming weeks, the overall structure will remain the same.

**Main components.**

- `train_sft.py` Implements supervised fine-tuning of the base model on instruction–response pairs. Supports full finetuning and LoRA-style finetuning.
- `train_pref.py` Implements preference optimization methods such as DPO. Takes an SFT checkpoint and a preference dataset.
- `train_rm.py` Trains a reward model on preference data (only needed for PPO/RLHF track).
- `eval.py` Runs automatic metrics, generation, and rubric-based evaluation on fixed prompt sets.
- `configs/` YAML configuration files for baseline experiments.
- `scripts/` Dataset preprocessing and experiment launch scripts.

## 3.1 Datasets and Evaluation Splits

**Instruction data (SFT).** This dataset consists of triples $(x, y)$ where:

- $x$ is an instruction or user prompt,
- $y$ is a high-quality reference response.

You will use:

- **Required:** Tulu 3/OLMo 2 SFT dataset from here: `https://huggingface.co/datasets/allenai/tulu-3-sft-olmo-2-mixture-0225`.
- **Optional:** additional instruction data with documented source and license of your choosing.

**Preference data.** This dataset consists of tuples $(x, y^+, y^-)$ where:

- $x$ is a prompt,
- $y^+$ is a preferred response,
- $y^-$ is a less preferred response.

Preferences encode helpfulness, safety, style, or instruction-following quality. You will train the model with DPO (direct preference optimization for the required component). You will use:

- **required**: OLMo 2 1B Preference Dataset `https://huggingface.co/datasets/allenai/olmo-2-0425-1b-preference-mix`
- **Optional**: additional instruction data with documented source and license of your choosing.

**Evaluation sets.** We will provide four fixed evaluation sets:

- `General`: general instruction-following and helpfulness. Specifically, we will use IFEval.
- `Safety`: harmful, sensitive, and refusal-triggering prompts. We will use Tulu 3 Safety dataset.
- `Math`: formatting traps, long-context, adversarial prompts. We will use GSM-8K for this.
- `Coding`: generating code given natural language instructions. We will use MBPP for this.

You can learn more about these evaluation framework here: `https://github.com/allenai/olmes`. You must **never** train on any of these evaluation prompts.

## 3.2 Part A: Supervised Fine-Tuning (SFT)

In this stage, you convert the base pretrained model into an *instruction-following* model.

**Goal.** Teach the model to:

- follow instructions more reliably,
- produce longer and more structured answers,
- reduce generic or degenerate completions.

## 3.3 Part B: Alignment with Preferences or RL

In this stage, you further shape behavior using user preferences or reward signals. In the default setup you will improve direct preference optimization, i.e., you will directly optimize the model so that preferred responses receive higher probability than dispreferred ones.

**Requirements.**

- Start from your best SFT checkpoint.
- Compare **SFT vs. SFT+Preference** on all evaluation sets.

### 3.3.1 Evaluation and Tradeoffs

After DPO, you must carefully analyze:

- improvements in instruction-following,
- changes in refusal behavior,
- degradation in factuality or diversity,
- robustness to adversarial prompts.

Your report should explore:

> How does alignment change behavior relative to SFT, and what new failure modes does it introduce?

### 3.4 Extensions and Explorations

After completing all baseline requirements, you must choose **1–2 extensions** from the list below *or propose your own extension* (subject to approval in your project proposal). The purpose of extensions is to encourage deeper exploration, creativity, and analysis beyond the default pipeline.

**Important grading note.** A correct and well-analyzed baseline implementation (SFT + alignment + evaluation) is sufficient to earn approximately **70% of the project grade**. Extensions are an opportunity to:

- earn additional credit,
- differentiate strong projects,
- explore directions closer to current research or real-world systems.

Extensions will be evaluated primarily on the correctness of implementation as well as insights and analysis, *not* on whether they improve the target performance.

Below are example extension categories with concrete suggestions. These are not exhaustive and will be updated with more suggestions in the new couple weeks.

#### 3.4.1 Data-Centric Extensions

- **Data filtering and quality analysis.** Analyze instruction or preference data quality (length, toxicity, duplication, style). Compare training on filtered vs. unfiltered data and discuss behavior changes.
- **Curriculum learning.** Train the model using a curriculum (e.g., short $\rightarrow$ long instructions, simple $\rightarrow$ complex prompts) and analyze convergence and stability.
- **Hard-negative or contrastive preference construction.** Generate or mine more challenging preference pairs (e.g., near-ties, subtle safety violations) and study their impact on alignment.
- **Preference noise analysis.** Inject controlled noise into preference labels and study robustness of DPO / RLHF to imperfect feedback.

#### 3.4.2 Alignment and Training Method Extensions

- **RL with verifiable rewards (RLVR).** Design a reward function based on automatically verifiable criteria (e.g., exact-match correctness, formatting constraints, unit tests for simple programs) and analyze how it compares to preference-based alignment.

- **Comparison of alignment methods.** Compare two alignment approaches (e.g., DPO vs. IPO, DPO vs. PPO) under the same compute budget and analyze stability, sample efficiency, and behavior differences.
- **Reward shaping or auxiliary objectives.** Augment the alignment objective with penalties or bonuses (e.g., verbosity control, refusal consistency) and analyze tradeoffs.
- **KL control and over-alignment analysis.** Study how changing KL targets or $\beta$ values affects refusal rates, diversity, and loss of capabilities.

#### 3.4.3 Evaluation and Analysis Extensions

- **Safety vs. helpfulness tradeoff curves.** Explicitly quantify tradeoffs by sweeping alignment strength and plotting refusal rate vs. task performance.
- **Calibration and uncertainty estimation.** Analyze whether aligned models are better calibrated (e.g., likelihood vs. correctness, abstention behavior).
- **Targeted error analysis.** Perform a structured qualitative analysis of failure modes (hallucination, over-refusal, verbosity collapse) with categorized examples.
- **Behavioral regressions.** Identify capabilities that degrade after alignment (e.g., reasoning length, creativity, formatting flexibility).

#### 3.4.4 Coding, Reasoning, and Task-Specific Extensions

- **Code generation and verification.** Evaluate models on a small coding task (e.g., Python functions) with automatic unit tests and analyze correctness vs. verbosity.
- **Reasoning and structure.** Study how alignment affects chain-of-thought length, structure, or consistency (without exposing hidden reasoning in the final output).
- **Format-constrained generation.** Align the model to follow strict output formats (JSON, XML, markdown tables) and evaluate compliance.

#### 3.4.5 Systems and Efficiency Extensions

- **Inference efficiency.** Measure latency, throughput, or token usage before and after alignment. Analyze whether alignment affects generation length or decoding cost.
- **Distillation.** Distill an aligned model into a smaller student model and analyze performance retention.
- **PEFT design analysis.** Study the effect of LoRA rank, layer placement, or freezing strategies on alignment effectiveness and efficiency.

#### 3.4.6 Open-Ended Proposals

You are encouraged to propose your own extension ideas, including:

- novel evaluation protocols,
- domain-specific alignment (e.g., medical, legal, educational prompts),
- human-in-the-loop evaluation or annotation,
- interpretability or representation analysis.

Proposed extensions should clearly state:

- the motivation,
- the experimental setup,
- the evaluation criteria,
- the expected insights.

**Evaluation of extensions.** Extensions are graded on:

- clarity of motivation,
- soundness of experimental design,
- depth of analysis and interpretation,
- honest discussion of negative or null results.

Performance improvements are *not* required for full credit.
You are allowed to use Tinker or OSC for any of these extensions.

### 3.5 Reproducibility and Submission Package

Alongside the final project report, you must submit a single `.zip` file containing:

- all training and evaluation code,
- configuration files and command lines,
- random seeds and environment description,
- a short model card describing training data, use cases, and limitations,
- final metrics and qualitative examples.

We will run your reproduction script on a subset of experiments.

### 3.6 Grading Criteria

Grades are based on:

- correctness and completeness of implementation,
- quality of experimental design,
- depth of analysis and interpretation,
- reproducibility and documentation,
- clarity of writing and presentation.

We are also currently working on creating a leaderboard for the default project. Will share more details in the upcoming week.

### 3.7 Academic Integrity

- You may discuss ideas with other teams but may not share code or trained models.
- You may use AI assistive tools for ideation, writing parts of the codebase/debugging your code, help with improving writing, but not use it as a tool to replace your work. Extensive usage of AI is discouraged.
- You must cite all external datasets, libraries, and resources.
- Any form of data leakage or result fabrication will result in course failure.