Final Project: RLHF for Open-Ended Instruction Following

Milestone Report Due April 13, 11:59 PM
Final Report Due May 13, 11:59 PM

AI Policy

You may *not* use AI assistants (chatbots, coding agents, etc.) to directly generate any material that will be graded. This includes both code and report text. You may use AI tools the same way you would use a human collaborator for limited help with debugging, explaining a concept, or pointing out a possible mistake, but you may not ask them to write your project for you or give them control of your terminal.

This is the same policy used elsewhere in the course: use AI as a limited aid, not as a substitute for your own implementation, experimental design, or writing.

1 Introduction

In this default final project, you will improve a small open-weight instruction-following language model using reinforcement learning from human feedback (RLHF). The base model is `Qwen/Qwen2.5-1.5B-Instruct`. The benchmark consists of open-ended user prompts, and the goal is to produce responses that are preferred to the frozen base model's responses.

This project has two parts:

1. **Part 1: Required implementation.** You will implement a fixed set of offline and online RLHF methods, train them on a fixed benchmark, and analyze their behavior.
2. **Part 2: Open-ended investigation.** You will design and carry out your own investigation into methods that improve performance on the same benchmark. This part asks you to choose promising ideas, implement them carefully, and evaluate them rigorously.

Contents

2 Project Setup

The repository contains the following important pieces:

- `llm_rl_final_proj/`: the main training and evaluation package,
- `scripts/modal_train.py`: Modal entrypoints,
- `dataset/wildchat_min4_judged_5k_v1/`: the benchmark dataset,
- `public_eval/`: the repository evaluation assets used for GPT-5.4 evaluation and Gradescope submissions.

The released codebase for this project is:

https://github.com/berkeleydeeprlcourse/homework_spring2026/tree/main/final_project_llm_rl

Initial setup:

```
uv run modal token new
uvx wandb login
```

For local GPT-5.4 evaluation, you will also need your own OpenAI API key in your shell environment:

```
export OPENAI_API_KEY=...
```

You do *not* need the OpenAI API for training. You need it only when you want to run the repository evaluation locally: your model generates responses on the provided prompt files, the frozen base model generates its own responses, and GPT-5.4 is then asked which response is better for each prompt. You should assume that you are responsible for the small API cost of these local evaluations. We require students to use their own OpenAI keys for this development loop so that you can evaluate freely without depending on the capped number of Gradescope submissions.

At current GPT-5.4 pricing,[1] one full local policy evaluation on the provided `128`-prompt evaluation file costs about **$1.07**. The exact cost will vary with response length.

Before running any training jobs on Modal, upload the dataset to your Modal volume:

```
uv run modal volume put llm-rl-final-project-volume dataset/wildchat_min4_judged_5k_v1 /
    synthetic_datasets/
```

You can verify the upload with:

[1] OpenAI API pricing.

```
uv run modal volume ls llm-rl-final-project-volume /synthetic_datasets
```

The evaluation files already live in the repository under `public_eval/`, so you do not need to upload them to your Modal volume. Inside Modal, use paths under `/root/project/public_eval/`.

Compute budget. Your total budget on Modal for the course is **$500**. This final project should fit comfortably within that budget for most groups. If you work in a team, you may pool your Modal resources. We are also looking into the possibility of providing additional credits, but you should not assume those will be available.

GPU choice. The Part 1 reference runs are designed to work with the standard GPU setup used in the provided commands. For some heavier Part 2 investigations, you may find that your method needs more memory. In that case, it is acceptable to switch to an `H200` Modal entrypoint. Treat this as an option for memory-intensive experiments, not as the default starting point.

3 Benchmark Dataset

The benchmark dataset has four splits:

- `train_prefs`: preference pairs for offline preference optimization and reward-model training,
- `test_prefs`: held-out preference pairs for reward-model and offline diagnostics,
- `train_gen`: prompt-only split for online RLHF rollouts,
- `test_gen`: held-out prompt-only split for policy evaluation.

Split sizes:

- `train_prefs`: 4744
- `test_prefs`: 256
- `train_gen`: 4744
- `test_gen`: 256

The prompt distributions for `train_prefs` and `train_gen` are the same. The difference is that `train_prefs` also contains paired responses and preference labels, while `train_gen` is the prompt source for online rollouts.

Throughout the project:

- offline methods use `train_prefs`,
- the reward model is trained on `train_prefs`,
- online methods use `train_gen` plus a reward model trained on `train_prefs`,
- held-out evaluation uses `test_prefs` and `test_gen`.

Before you start coding, open a few rows from each split and inspect them. You should understand what a prompt looks like, what a chosen/rejected pair looks like, and how the prompt-only generation splits differ from the preference-pair splits.

For example:

```
head -n 2 dataset/wildchat_min4_judged_5k_v1/train_prefs.jsonl
head -n 2 dataset/wildchat_min4_judged_5k_v1/train_gen.jsonl
```

4 Evaluation Protocol

4.1 Primary metric

The main score is **win rate against the frozen base model under an external LLM judge**. Given a held-out prompt x:

1. generate a response from your trained model,
2. generate a response from the frozen base model,
3. ask an external judge which response is better,
4. compute the fraction of evaluation prompts on which your model wins.

This project uses GPT-5.4 as the reference judge. The key point is that your policy is never scored in isolation; it is scored in a head-to-head comparison against the base model on the same prompts.

4.2 Evaluation assets

The repository includes evaluation prompt and preference files:

- `public_test_gen_prompts_128.jsonl`: policy evaluation prompt set used for both Part 1 and Part 2,
- `public_test_prefs_256.jsonl`: reward-model evaluation pairs.

Use your own OpenAI API key to run these evaluations during development. Official Gradescope submissions will be limited to **five total submissions**, so you should treat the provided evaluation files as your main feedback loop and Gradescope as a confirmation step.

4.3 Secondary diagnostics

You should also monitor:

- held-out preference accuracy on `test_prefs`,
- reward-model pair accuracy on `test_prefs`,
- approximate KL to the frozen reference,
- reward-model score improvements for online methods,
- generation-collapse diagnostics and qualitative sample quality.

These diagnostics matter because they help you debug why a method is helping or failing. They are not, however, the final metric. A method can look better according to internal metrics and still perform worse in the final head-to-head LLM evaluation.

5 Codebase Overview

In the starter code, the most important files are:

- `llm_rl_final_proj/models/logprobs.py`: token log-probabilities, completion masking, sampled KL,
- `llm_rl_final_proj/rollout/rollout_buffer.py`: rollout minibatching,
- `llm_rl_final_proj/offline/losses.py`: offline preference objectives,
- `llm_rl_final_proj/reward_model/train.py`: Bradley–Terry reward-model loss,
- `llm_rl_final_proj/rl/grpo.py`: GRPO update,
- `llm_rl_final_proj/rl/dr_grpo.py`: DrGRPO update,
- `llm_rl_final_proj/rl/gspo.py`: GSPO update,
- `llm_rl_final_proj/online/train_rm_grpo.py`: online rollout pipeline, advantage construction, and algorithm wiring,
- `llm_rl_final_proj/online/train_rm_online_pref.py`: extension hook for Part 2 online preference methods,
- `llm_rl_final_proj/online/train_rm_ppo.py`: extension hook for Part 2 PPO-style methods.

The starter code intentionally leaves the mathematically important parts unimplemented. Your job in Part 1 is to complete those TODOs and then use the same codebase as a platform for Part 2 extensions.

6 Reading Before You Start

You should read the original method papers while you work through Part 1. The code becomes much easier to understand once you know what the objective is trying to do.

- DPO: *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*
- IPO: *A General Theoretical Paradigm to Understand Learning from Human Preferences*
- AOT: *Distributional Preference Alignment via Optimal Transport*
- Reward modeling in RLHF: *Training language models to follow instructions with human feedback*
- GRPO: *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*
- DrGRPO: *DrGRPO*
- GSPO: *GSPO*

You do not need to master every detail before writing code, but you should know the high-level motivation, the main loss, and what changes from one method to the next.

7 Part 1 Theory

7.1 Notation

Let:

- x denote a prompt,
- y denote a full response sequence,
- y^+ and y^- denote chosen and rejected responses,
- π_θ denote the current policy,
- π_{ref} denote the frozen reference policy,
- $r_\phi(x, y)$ denote the learned reward model score,
- $\beta > 0$ denote the preference-loss scale parameter.

For a full response sequence, define the reference-corrected preference margin

$$\Delta_\theta(x, y^+, y^-) = \big(\log \pi_\theta(y^+ \mid x) - \log \pi_\theta(y^- \mid x)\big) - \big(\log \pi_{\text{ref}}(y^+ \mid x) - \log \pi_{\text{ref}}(y^- \mid x)\big).$$

This margin asks: compared with the frozen base model, how much more does the current policy prefer the chosen response over the rejected response?

7.2 Offline methods

7.2.1 DPO

Direct Preference Optimization uses the logistic loss

$$\mathcal{L}_{\text{DPO}} = -\log \sigma\big(\beta\, \Delta_\theta(x, y^+, y^-)\big).$$

Interpretation:

- If the policy already gives the chosen response a larger reference-corrected score than the rejected response, then Δ_θ is positive and the loss is small.
- If the policy prefers the rejected response, then Δ_θ is negative and the loss becomes large.
- The scalar β controls how aggressively the objective responds to changes in the preference margin.

DPO is attractive because it turns preference learning into a direct policy-optimization problem. There is no separate reward model in the offline branch: the policy and reference together define the training signal.

7.2.2 IPO

Implicit Preference Optimization uses the same reference-corrected margin, but instead of applying a logistic loss, it matches the margin to a target value:

$$\mathcal{L}_{\text{IPO}} = \left(\Delta_\theta(x, y^+, y^-) - \frac{1}{2\beta}\right)^2.$$

Intuition:

- DPO keeps pushing to separate chosen and rejected responses through a logistic objective.
- IPO instead treats the desired preference margin as a regression target.
- This often makes IPO feel more conservative than DPO: rather than endlessly increasing margins, it aims for a particular scale.

IPO is therefore a useful comparison point because it changes the shape of the loss while keeping almost the same ingredients as DPO.

7.2.3 AOT

Alignment via Optimal Transport (AOT) changes the granularity of comparison. Instead of only optimizing each chosen/rejected pair independently, it compares the *distribution* of chosen scores against the distribution of rejected scores across a minibatch.

In this codebase, each example produces a scalar reference-corrected sequence reward. AOT sorts the chosen rewards and the rejected rewards separately and then applies a DPO-style penalty to the sorted quantiles:

$$\mathcal{L}_{\text{AOT}} = \frac{1}{B}\sum_{i=1}^{B} -\log\sigma\big(\beta(r^{+}_{(i)} - r^{-}_{(i)})\big).$$

Here $r^{+}_{(i)}$ is the i-th sorted chosen reward and $r^{-}_{(i)}$ is the i-th sorted rejected reward. Intuitively, AOT encourages the entire chosen distribution to shift upward relative to the rejected distribution, not only each individual paired example.

7.3 Reward modeling

The reward model takes a prompt-response pair (x, y) and produces a scalar score $r_\phi(x, y)$. It is trained with the Bradley–Terry objective

$$\mathcal{L}_{\text{RM}} = -\log\sigma\big(r_\phi(x, y^{+}) - r_\phi(x, y^{-})\big).$$

Why this loss makes sense:

- If the model scores the chosen response higher than the rejected response, the margin $r_\phi(x, y^{+}) - r_\phi(x, y^{-})$ is positive and the loss is small.
- If it scores them in the wrong order, the loss becomes large.
- The resulting reward model is only identifiable up to an additive constant, but that is fine because online RL uses reward *differences* and normalized advantages.

7.4 Online methods

For online RLHF, we sample groups of responses for each prompt, score them with the reward model, and then update the policy with a clipped policy-gradient objective.

7.4.1 Group-relative advantages

For one prompt x_i, suppose we sample G responses with rewards $r_{i,1}, \ldots, r_{i,G}$. GRPO-style grouped normalization forms advantages

$$A_{i,j} = \frac{r_{i,j} - \mu_i}{\sigma_i + \varepsilon}, \qquad \mu_i = \frac{1}{G}\sum_{j=1}^{G} r_{i,j}, \qquad \sigma_i = \sqrt{\frac{1}{G}\sum_{j=1}^{G}(r_{i,j} - \mu_i)^2}.$$

This produces one scalar advantage per sampled response. A response is rewarded when it is better than the other samples for the same prompt, and penalized when it is worse.

7.4.2 Sampled KL regularization

The online methods regularize the trainable policy toward the frozen reference policy. At one token position, define

$$\Delta(a) = \log \pi_{\text{ref}}(a \mid s) - \log \pi_\theta(a \mid s).$$

The code uses the sampled-token nonnegative KL proxy

$$\widehat{k}(a) = e^{\Delta(a)} - \Delta(a) - 1.$$

This quantity has two useful properties:

- its expectation under $a \sim \pi_\theta(\cdot \mid s)$ equals the exact token-level KL,
- each sampled token contributes a nonnegative penalty rather than a signed quantity that can flip negative.

In the implementation, this token-level proxy is averaged over completion tokens and multiplied by a coefficient `kl_coef`.

7.4.3 GRPO

GRPO uses a PPO-style clipped surrogate at the token level. Let

$$\rho_{i,t} = \frac{\pi_\theta(y_{i,t} \mid x_i, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x_i, y_{i,<t})}.$$

For a response with scalar advantage A_i, the surrogate compares the unclipped and clipped terms

$$\rho_{i,t} A_i \qquad \text{and} \qquad \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon) A_i.$$

In this project's GRPO implementation, those token-level terms are averaged over the completion tokens in the response, which removes an otherwise undesirable bias toward longer responses.

7.4.4 DrGRPO

DrGRPO changes two things relative to GRPO:

1. it uses mean-centered rewards without dividing by the per-group standard deviation,

2. it removes the per-sequence length normalization from the surrogate.

The motivation is that these changes alter how GRPO handles sequence length and reward scaling. In particular, removing the per-sequence normalization changes the length dependence of the update, which makes DrGRPO a useful comparison point when you are thinking about length bias in policy optimization.

7.4.5 GSPO

GSPO changes where clipping is applied. Instead of clipping per-token likelihood ratios, it aggregates token log-ratios into a single sequence-level ratio and then applies PPO-style clipping at the sequence level.

Conceptually:

- GRPO clips locally at each token,
- GSPO clips globally at the response level.

This changes the optimization geometry: GSPO treats the response as one structured action rather than a bag of token-wise policy updates.

8 Part 1: Required Implementation

You must implement the following pieces in the starter code:

- **Log-probability utilities:**
  - `compute_per_token_logprobs`
  - `build_completion_mask`
  - `approx_kl_from_logprobs`

  in `llm_rl_final_proj/models/logprobs.py`
- **Rollout minibatching:**
  - `iter_minibatches`

  in `llm_rl_final_proj/rollout/rollout_buffer.py`
- **Offline losses:**
  - DPO, IPO, and AOT branches

  in `llm_rl_final_proj/offline/losses.py`
- **Reward-model training:**
  - Bradley–Terry margin and loss

  in `llm_rl_final_proj/reward_model/train.py`
- **Online RL updates:**
  - GRPO

  - DrGRPO
  - GSPO

  in `llm_rl_final_proj/rl/grpo.py`, `llm_rl_final_proj/rl/dr_grpo.py`, and `llm_rl_final_proj/rl/gspo.py`

- **Online advantage construction and wiring:**
  - grouped-advantage computation in `llm_rl_final_proj/online/train_rm_grpo.py`
  - the algorithm-specific choice of whether to divide advantages by the group standard deviation in `llm_rl_final_proj/online/train_rm_grpo.py`

Part 1 required methods. You must train and evaluate:

- one Bradley–Terry reward model,
- three offline methods: DPO, IPO, AOT,
- three online methods: GRPO, DrGRPO, GSPO.

9 Part 1: Required Commands

Use the following commands as your starting point for Part 1. They are the configurations we recommend you run first after you finish the required implementations. Once these are working, you can use W&B, the provided evaluation files, and your own experiments to decide what to change in Part 2.

9.1 Reward model

```
uv run modal run scripts/modal_train.py::reward_model_train_remote -- \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_prefs \
  --eval_split test_prefs \
  --output_dir /vol/runs/wildchat_min4_judged_5k_reward_model_v1 \
  --per_device_train_batch_size 8 \
  --per_device_eval_batch_size 8 \
  --grad_accum_steps 4 \
  --lr 3e-5 \
  --num_train_epochs 3 \
  --max_prompt_tokens 700 \
  --max_response_tokens 512 \
  --eval_interval 25 \
  --save_interval 50 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_reward_model_v1
```

9.2 Offline methods

DPO

```
uv run modal run scripts/modal_train.py::train_remote -- \
```

```
  --algo dpo \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_prefs \
  --eval_split test_prefs \
  --generation_split test_gen \
  --output_dir /vol/runs/wildchat_min4_judged_5k_dpo_beta005_v1 \
  --beta 0.005 \
  --per_device_train_batch_size 4 \
  --per_device_eval_batch_size 4 \
  --grad_accum_steps 4 \
  --lr 5e-5 \
  --num_train_epochs 3 \
  --max_prompt_tokens 700 \
  --max_response_tokens 512 \
  --generation_eval_limit 32 \
  --generation_eval_max_new_tokens 256 \
  --generation_eval_every 100 \
  --eval_interval 100 \
  --save_interval 100 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_dpo_beta005_v1
```

IPO

```
uv run modal run scripts/modal_train.py::train_remote -- \
  --algo ipo \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_prefs \
  --eval_split test_prefs \
  --generation_split test_gen \
  --output_dir /vol/runs/wildchat_min4_judged_5k_ipo_v1 \
  --beta 0.1 \
  --per_device_train_batch_size 4 \
  --per_device_eval_batch_size 4 \
  --grad_accum_steps 4 \
  --lr 5e-5 \
  --num_train_epochs 3 \
  --max_prompt_tokens 700 \
  --max_response_tokens 512 \
  --generation_eval_limit 32 \
  --generation_eval_max_new_tokens 256 \
  --generation_eval_every 100 \
  --eval_interval 100 \
  --save_interval 100 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_ipo_v1
```

AOT

```
uv run modal run scripts/modal_train.py::train_remote -- \
  --algo aot \
```

```
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_prefs \
  --eval_split test_prefs \
  --generation_split test_gen \
  --output_dir /vol/runs/wildchat_min4_judged_5k_aot_beta02_v1 \
  --beta 0.2 \
  --per_device_train_batch_size 4 \
  --per_device_eval_batch_size 4 \
  --grad_accum_steps 4 \
  --lr 5e-5 \
  --num_train_epochs 3 \
  --max_prompt_tokens 700 \
  --max_response_tokens 512 \
  --generation_eval_limit 32 \
  --generation_eval_max_new_tokens 256 \
  --generation_eval_every 50 \
  --eval_interval 50 \
  --save_interval 50 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_aot_beta02_v1
```

9.3 Online methods

First train the reward model. Then run each online algorithm with the saved reward-model adapter.

GRPO

```
uv run modal run scripts/modal_train.py::rm_grpo_train_remote -- \
  --algo grpo \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_gen \
  --eval_split test_gen \
  --reward_model_name Qwen/Qwen2.5-1.5B-Instruct \
  --reward_adapter_path /vol/runs/wildchat_min4_judged_5k_reward_model_v1/checkpoints/
    step_000200/adapter \
  --output_dir /vol/runs/wildchat_min4_judged_5k_grpo_v1 \
  --steps 25 \
  --batch_size 16 \
  --group_size 4 \
  --min_new_tokens 32 \
  --max_new_tokens 256 \
  --temperature 0.8 \
  --top_p 0.95 \
  --lr 1e-5 \
  --grad_accum_steps 2 \
  --ppo_epochs 2 \
  --minibatch_size 8 \
  --clip_eps 0.2 \
  --kl_coef 0.01 \
  --max_prompt_tokens 700 \
  --max_response_tokens 256 \
```

```
  --eval_limit 32 \
  --eval_interval 25 \
  --save_interval 25 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_grpo_v1
```

DrGRPO

```
uv run modal run scripts/modal_train.py::rm_grpo_train_remote -- \
  --algo dr_grpo \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_gen \
  --eval_split test_gen \
  --reward_model_name Qwen/Qwen2.5-1.5B-Instruct \
  --reward_adapter_path /vol/runs/wildchat_min4_judged_5k_reward_model_v1/checkpoints/
    step_000200/adapter \
  --output_dir /vol/runs/wildchat_min4_judged_5k_drgrpo_v1 \
  --steps 25 \
  --batch_size 16 \
  --group_size 4 \
  --min_new_tokens 32 \
  --max_new_tokens 256 \
  --temperature 0.8 \
  --top_p 0.95 \
  --lr 1e-5 \
  --grad_accum_steps 2 \
  --ppo_epochs 2 \
  --minibatch_size 8 \
  --clip_eps 0.2 \
  --kl_coef 0.01 \
  --max_prompt_tokens 700 \
  --max_response_tokens 256 \
  --eval_limit 32 \
  --eval_interval 25 \
  --save_interval 25 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_drgrpo_v1
```

GSPO

```
uv run modal run scripts/modal_train.py::rm_grpo_train_remote -- \
  --algo gspo \
  --model_name Qwen/Qwen2.5-1.5B-Instruct \
  --dataset_name /vol/synthetic_datasets/wildchat_min4_judged_5k_v1 \
  --train_split train_gen \
  --eval_split test_gen \
  --reward_model_name Qwen/Qwen2.5-1.5B-Instruct \
  --reward_adapter_path /vol/runs/wildchat_min4_judged_5k_reward_model_v1/checkpoints/
    step_000200/adapter \
  --output_dir /vol/runs/wildchat_min4_judged_5k_gspo_v1 \
  --steps 25 \
  --batch_size 16 \
```

```
  --group_size 4 \
  --min_new_tokens 32 \
  --max_new_tokens 256 \
  --temperature 0.8 \
  --top_p 0.95 \
  --lr 1e-5 \
  --grad_accum_steps 2 \
  --ppo_epochs 2 \
  --minibatch_size 8 \
  --clip_eps 0.2 \
  --kl_coef 0.01 \
  --max_prompt_tokens 700 \
  --max_response_tokens 256 \
  --eval_limit 32 \
  --eval_interval 25 \
  --save_interval 25 \
  --wandb_enabled \
  --wandb_project llm-rl-final-project \
  --wandb_name wildchat_min4_judged_5k_gspo_v1
```

10 W&B and Debugging Advice

The codebase logs generations to W&B during both offline and online training. You should inspect:

- the Markdown sample panel,
- the `samples/generation_table` table,
- held-out preference metrics,
- online KL and reward-model score metrics,
- generation-collapse diagnostics.

A method is *not* correct just because the loss decreases. Internal metrics can improve even when response quality gets worse according to the final head-to-head LLM evaluation. You should regularly inspect the model's generations while you debug and compare methods.

Some useful debugging questions:

- Do chosen responses actually get larger policy scores than rejected responses?
- Does your reward model assign higher scores to chosen responses than rejected responses?
- Do your online methods produce diverse, on-topic responses, or do they drift into repetition and generic filler?
- When a method underperforms, is the problem the objective, the reward model, or the checkpoint you are evaluating?

11 Part 2: Open-Ended Investigation

In Part 2, you will try to improve benchmark performance beyond the Part 1 baselines. You may work on offline methods, online methods, or both. Your job is to identify promising ideas, turn

them into concrete code changes, and then test those changes carefully.

Important constraint. Part 2 is about improving the methods, not about giving yourself more data. You should continue to use the same benchmark dataset, the same split structure, and the same evaluation files. Do not train on extra preference data or extra prompt data beyond what is provided in the repository.

Keep the benchmark fixed. For both Part 1 and Part 2, do not change the base model, the dataset splits, the evaluation files, or the token budgets used by the benchmark. In particular, keep `max_prompt_tokens=700`. For offline training and reward-model training, keep `max_response_tokens=512`. For online training and submission generation, keep `max_new_tokens=256` and `max_response_tokens=256`. Also use deterministic decoding (`temperature=0.0`, `top_p=1.0`) when generating the JSONL files that you submit. If a memory-heavy Part 2 idea does not fit on the default GPU, you may switch to an `H200` entrypoint, but you should not change these benchmark settings just to fit smaller hardware.

11.1 How to approach Part 2

Good Part 2 work usually looks like this:

1. identify a mechanism that might plausibly improve head-to-head win rate against the base model,
2. implement that mechanism by modifying an existing trainer rather than writing a huge new system from scratch,
3. compare it against the relevant Part 1 baseline under the same benchmark,
4. inspect both quantitative metrics and qualitative samples,
5. keep what works and discard what does not.

The intended extension points are:

- `llm_rl_final_proj/offline/losses.py` for new offline objectives,
- `llm_rl_final_proj/online/train_rm_grpo.py` for GRPO-family variants,
- `llm_rl_final_proj/online/train_rm_online_pref.py` for replay or online-preference methods,
- `llm_rl_final_proj/online/train_rm_ppo.py` for PPO-style actor-critic methods,
- `llm_rl_final_proj/config.py` if your new method needs additional knobs.

You should not need to write hundreds of lines of unrelated infrastructure. A strong Part 2 project usually comes from changing the right objective, baseline, aggregation rule, reward-model usage pattern, or checkpoint-selection strategy.

11.2 Possible directions

The list below is meant to help you think about what to try. You should treat these as starting points for your own investigation, not as a script to follow mechanically.

11.2.1 Offline ideas

Confidence-aware pair weighting. Not all preference pairs are equally informative. If you can identify pairs where the chosen/rejected distinction is especially clear, one idea is to weight those pairs more heavily in the objective while still keeping a floor on weaker pairs so they are not discarded entirely.

Push–pull objectives that treat good and bad responses asymmetrically. A standard DPO-style loss compares chosen and rejected responses through one margin. Another family of ideas treats "push the chosen response up" and "push the rejected response down" as distinct effects, often with different nonlinearities. This can make the objective more robust to noisy pairs or to already-strong chosen responses. The APO family is an example of this idea.

Reference-free max-margin objectives. The reference model is useful, but it is not mathematically required for every preference objective. One alternative is to train a pure margin-based policy objective that prefers chosen responses over rejected responses directly and controls scale by other means.

Other natural offline variations. A few other directions are worth thinking about:

- objectives that weight examples by the current policy distribution, as in the WPO family,
- objectives that replace hard pairwise labels with a softer target or label smoothing,
- objectives that try to align with binary desirability rather than paired comparison margins, as in KTO-style approaches.

11.2.2 Online ideas

Reward-model checkpoint selection. The best reward-model checkpoint for downstream policy improvement is not always the last one. One straightforward but powerful idea is to compare different reward-model checkpoints inside the same online training recipe.

Reward-model ensembles and pessimistic aggregation. If you have multiple reasonable reward-model checkpoints, one can aggregate them in ways that reward responses only when they look good to all models or when the reward estimate is high with low uncertainty. This is a natural way to reduce reward hacking.

Alternative baselines and control variates. Policy-gradient methods can become much better or much worse depending on the baseline. Useful ideas include leave-one-out baselines, rank-only baselines, greedy-response baselines, and other advantage-normalization schemes. This includes

simple changes to how advantages are centered or how a strong baseline response is chosen, and it also points toward ReMax-style greedy baselines.

Replay or online-preference methods. Instead of discarding all previous rollouts immediately, one can keep a small replay set and train an online preference objective on the resulting data. This is especially interesting when you want to compare strict on-policy updates against methods that reuse recent rollouts, such as online DPO- or IPO-style updates.

Additional online variations. Some other algorithmically interesting directions are:

- actor-critic methods with a learned value function and GAE,
- reward centering or reward calibration schemes,
- rank-only or geometry-changing GRPO variants, including GOPO-style ideas.

A good Part 2 project does not need all of these. A small number of well-chosen ideas, combined with careful empirical comparisons, is usually much stronger than a long list of shallow experiments.

If you want a literature starting point for Part 2, useful search terms include *KTO*, *APO*, *WPO*, *RePO*, *ReMax*, *GOPO*, reward-model ensembles, and replay-based online preference optimization. You should read the relevant papers and decide for yourself which ideas are worth implementing on this benchmark.

12 Report Instructions

Submit a PDF report written in a short research-paper style. Your report should be understandable without reading your code.

12.1 Recommended report structure

Your report does not need to be long, but it should read like a concise empirical paper rather than a collection of screenshots. A good structure is:

- **Abstract.** One short paragraph stating what you implemented, what you investigated in Part 2, and your main findings.
- **Introduction.** Briefly describe the benchmark, the overall RLHF setting, and the questions you study.
- **Methods.** Explain the Part 1 methods and the Part 2 ideas you chose to investigate. Include equations where they matter.
- **Experimental setup.** Describe the dataset splits, model, hyperparameters, evaluation files, and any important implementation choices.
- **Results.** Present quantitative comparisons and qualitative examples. Tables should make it easy to compare Part 2 methods against the relevant Part 1 baselines.
- **Discussion.** Summarize what worked, what failed, what you learned, and what you would try next.

Your report should make it easy for the reader to answer three questions:

1. What did you implement?
2. What did you try in Part 2, and why?
3. What evidence supports your conclusions?

At a minimum, include:

1. **Problem setup.** Briefly describe the benchmark, the base model, and the evaluation protocol.
2. **Part 1 methods.** Explain the required offline and online methods and summarize what you observed.
3. **Part 2 methods.** Describe the methods or ideas you chose to investigate and motivate why they might help.
4. **Experimental setup.** State exactly what models, splits, hyperparameters, and evaluation procedures you used.
5. **Results.** Include tables and plots. Make it easy to compare your Part 2 methods against the Part 1 baselines.
6. **Analysis.** Explain what helped, what failed, and what you learned.

12.2 Questions to answer in your report

Part 1 questions

1. Why do DPO, IPO, and AOT optimize different notions of improvement even though all three use chosen/rejected preference pairs?
2. Which offline Part 1 method worked best for you, and what did you see in its samples that helped explain the result?
3. How did the reward model behave on held-out `test_prefs`? Did pair accuracy and margin quality line up with downstream policy quality?
4. Compare GRPO, DrGRPO, and GSPO. Which one worked best for you, and what do you think explains the differences?
5. Give one example where an internal metric looked encouraging but the actual generations or the head-to-head LLM evaluation suggested a problem.

Part 2 questions

1. What ideas did you try, and why did you think they might help on this benchmark?
2. Which changes actually improved win rate under the external LLM judge? Were those improvements coming from the objective itself, from reward-model usage, from checkpoint choice, or from optimization details?
3. Which ideas failed, and what evidence convinced you that they failed?
4. If your strongest Part 2 method was offline, explain why it beat the Part 1 offline baselines. If it was online, explain why it beat the Part 1 online baselines.

5. If you had twice as much experimentation budget, what would you test next and why?

13 Grading Rubric

Your final project grade is based on four components:

- **Part 1 implementation correctness: 20%**
- **Part 1 required experiments and artifacts: 17.5%**
- **Part 2 investigation quality: 37.5%**
- **Final report: 25%**

1. Part 1 implementation correctness

We will check whether your implementations of:

- DPO,
- IPO,
- AOT,
- reward-model training,
- GRPO,
- DrGRPO,
- GSPO

are correct and produce the expected behavior on the provided benchmark.

2. Part 1 required experiments

We will check that you ran the required methods, logged them properly, and can discuss their behavior clearly in the report.

Code rubric

Category	Poor (0–59%)	Fair (60–79%)	Good (80–89%)	Excellent (90–100%)
Part 1 correctness	Major parts of the required algorithms are missing or incorrect; training runs do not produce meaningful outputs.	Some required components work, but there are still major implementation bugs, missing pieces, or unstable runs.	Required algorithms are implemented and train successfully, with only minor issues or weaker analysis of failure cases.	Required algorithms are implemented correctly, stable, and clearly understood; implementation choices are defended in the report.
Experimental completeness	Several required runs are missing, broken, or not comparable.	Most required runs are present, but the experimental record is incomplete, inconsistent, or hard to interpret.	Required runs are present and mostly comparable, with enough information to interpret them.	All required runs are complete, cleanly organized, and compared fairly using the same benchmark and evaluation pipeline.
Part 2 technical quality	Part 2 changes are shallow, under-tested, or not really connected to the methods.	Part 2 includes some real effort, but the methods, comparisons, or conclusions are still limited.	Part 2 includes at least one meaningful extension and a reasonable experimental comparison.	Part 2 demonstrates thoughtful algorithmic modification, careful implementation, and strong empirical discipline.

Gradescope performance thresholds

Gradescope uses the same evaluation files that you should use locally during development. The thresholds below are set slightly below the strongest reference runs on those same files. For Part 2, you receive full credit on that portion if you clear *either* the offline threshold or the online threshold.

Item	Autograder threshold
Reward model pair accuracy	0.74
DPO	0.68
IPO	0.55
AOT	0.72
GRPO	0.68
DrGRPO	0.64
GSPO	0.60
Part 2 offline best	0.83
Part 2 online best	0.75

3. Part 2 investigation quality

We will evaluate whether your Part 2 investigation is thoughtful and empirical. Strong submissions:

- try multiple plausible ideas,
- compare against the right baselines,
- beat at least one relevant Part 1 baseline in the regime you investigate,
- explain both successes and failures.

4. Final report quality

We will look for:

- correctness of method descriptions,
- clarity of writing,
- fair and controlled comparisons,
- good use of plots, tables, and qualitative samples,
- evidence that you understand the connection between the math and the empirical behavior.

Report rubric

Category	Poor (0–59%)	Fair (60–79%)	Good (80–89%)	Excellent (90–100%)
Method explanation	Descriptions are vague, incomplete, or mathematically incorrect.	The report covers the main methods, but some explanations are incomplete, thin, or occasionally unclear.	Methods are described correctly, though some motivation or details may be thin.	Methods are described clearly and correctly, with enough mathematical and algorithmic detail that the reader can follow the design choices.
Empirical analysis	Results are shown but not interpreted, or the comparisons are not fair.	The report contains useful results, but the comparisons or conclusions are only partially convincing.	Results are compared sensibly and the main conclusions are supported.	Results are compared carefully, failure cases are discussed honestly, and conclusions are clearly supported by both quantitative and qualitative evidence.
Communication quality	The report is hard to follow, missing important details, or poorly organized.	The report is readable, but organization or completeness still needs work.	The report is readable and mostly complete.	The report is well organized, concise, easy to follow, and answers the required questions directly.

14 Submission

There are **three** separate Gradescope assignments for this project:

1. **Artifacts submission (autograded).**
2. **Code submission (not autograded).**
3. **Report submission (not autograded).**

1. Artifacts submission (autograded)

This is the only Gradescope assignment that is autograded. You will upload a directory of JSONL files built from your trained models.

For Part 1, the artifacts submission must include:

- one reward-model JSONL on `public_test_prefs_256.jsonl`,
- six policy JSONLs on `public_test_gen_prompts_128.jsonl`.

For Part 2, use the same `128`-prompt evaluation file. Submit your strongest offline policy as `offline_best.jsonl`, your strongest online policy as `online_best.jsonl`, or both. Full credit for the Part 2 autograder portion is awarded if either the offline threshold or the online threshold is cleared.

- `offline_best.jsonl` on `public_test_gen_prompts_128.jsonl`,
- `online_best.jsonl` on `public_test_gen_prompts_128.jsonl`.

Use the README as the command reference for:

- uploading the dataset to Modal,
- building evaluation policy generations,
- building reward-model submission files,
- downloading and zipping the submission directory.

For debugging, you may run the local autograder on an incomplete submission directory. This is often useful when you want to check one method at a time. Missing files will simply be marked as failures.

2. Code submission (not autograded)

You must also submit your code as a separate Gradescope assignment. This submission is not autograded; it will be used for manual inspection and for reproducibility.

Your code submission should contain:

- your full project repository,
- all code needed to reproduce your Part 1 and Part 2 methods,
- a clearly visible top-level file or README section listing the exact commands needed to reproduce the runs you discuss in your report.

Those reproduction commands should be detailed enough that a course staff member could rerun your final experiments without guessing. In particular, include:

- the reward-model training command you used,
- the Part 1 commands you ran,
- the Part 2 commands for every method you discuss in the report,
- any important checkpoint paths, evaluation commands, and submission-building commands needed to verify your claims.

3. Report submission (not autograded)

Submit your final report PDF as a separate Gradescope assignment.

Practical advice. Treat Gradescope as a confirmation step, not as your main debugging loop. During development, use your own OpenAI key to run the provided head-to-head evaluation, inspect W&B generations frequently, and submit to Gradescope only when you already have a clear picture of what your model is doing. You will have at most **five** Gradescope submissions, so use them carefully.