Below is an extended project blueprint that starts with a conventional supervised fine-tuning (SFT) pipeline but can later be expanded to include more advanced techniques such as Preference Fine-Tuning, RLHF (Reinforcement Learning from Human Feedback), or RLAF (Reinforcement Learning from AI Feedback). This structure still follows the microservice or modular approach you’ve set up, so it’s easy to insert additional steps or services for reward modeling, preference data collection, and policy optimization (PPO or other RL algorithms).

1. Project Structure with Future RLHF in Mind

my\_fine\_tuning\_project/

├── data/

│ ├── raw/

│ ├── processed/

│ ├── preferences/ # Optional: Store preference data or reward-labeled data

│ └── README.md

├── docs/

│ ├── fine\_tuning\_instructions.md

│ └── rlhf\_instructions.md # A place to note advanced RLHF steps

├── microservices/

│ ├── data\_service/

│ │ ├── data\_loader.py

│ │ ├── data\_preprocessor.py

│ │ ├── preference\_data\_loader.py # For preference data (optional)

│ │ └── requirements.txt

│ ├── model\_service/

│ │ ├── model\_definition.py

│ │ ├── train.py

│ │ ├── infer.py

│ │ ├── reward\_model.py # (optional advanced) Code for the reward model

│ │ └── preference\_finetune.py # (optional advanced) For preference or RL-based steps

│ ├── orchestrator/

│ │ ├── pipeline.py

│ │ ├── config.py

│ │ ├── preference\_pipeline.py # (optional advanced) Orchestrate preference fine-tuning

│ │ └── rlhf\_pipeline.py # (optional advanced) Orchestrate RLHF steps

│ └── README.md

├── tests/

│ ├── test\_data\_service.py

│ ├── test\_model\_service.py

│ ├── test\_rlhf.py # (optional advanced) Tests for RL-based code

│ └── \_\_init\_\_.py

├── notebooks/

│ ├── exploration.ipynb

│ ├── prototype\_finetuning.ipynb

│ └── prototype\_preference\_rl.ipynb

├── .gitignore

├── README.md

└── main.py

Key additions for RLHF / RLAF / Preference Fine-Tuning:

A preferences/ subfolder under data/ to store human or AI-labeled data pairs (e.g., (prompt, answer1, answer2, preference)).

preference\_data\_loader.py in data\_service/ for reading and preparing preference data.

reward\_model.py in model\_service/ if you decide to train or fine-tune a separate reward model.

preference\_finetune.py or rlhf\_pipeline.py for orchestrating advanced steps (training the reward model, running PPO, etc.).

2. Starting with Supervised Fine-Tuning (SFT)

Why: SFT is often the first step, even in RLHF processes, because you need a good “base policy” before you refine it with reward signals or user feedback.

Data Ingestion (via data\_loader.py)

Preprocessing (via data\_preprocessor.py)

Model Definition (e.g., a transformer from Hugging Face or your own architecture)

Training (via train.py)

Inference (via infer.py)

Once the SFT model is at a decent quality, you can move on to preference fine-tuning or RLHF.

3. Preference Fine-Tuning

Also known as Instruction Tuning or SFT with Preference Data. The idea is that you have additional data that indicates how a user or crowd might rank multiple model outputs for the same input.

3.1 Collect or Load Preference Data

Data Format: For each prompt, you might have two or more candidate responses plus a label specifying which response is preferred. Something like:

{

"prompt": "Explain quantum entanglement in simple terms.",

"response\_1": "It’s about two particles that share states.",

"response\_2": "Entanglement is a phenomenon where...",

"preferred": 2

}

Loader: In preference\_data\_loader.py, parse these files into a structure that your model\_service can use.

3.2 Convert Preferences to Pairwise Examples

You can turn preference data into pairwise constraints (A vs. B) or produce a new “best single answer” dataset:

Pairwise: For training a reward model or direct rank-based approaches (e.g., sorting the likelihoods).

Single-Answer: If you just want to pick the best response to feed into your SFT model again.

3.3 Fine-Tune Your Model

You might feed the “winning answers” as ground truth to your model in a standard supervised way, or use a custom objective that accounts for the preference ranking.

4. Reinforcement Learning from Human Feedback (RLHF)

4.1 RLHF Pipeline Components

Supervised Fine-Tuned Model (Policy Init)

This is your starting policy (the SFT model).

Reward Model

A separate model that has been trained (often via supervised learning) to predict a “reward” given (prompt, response).

This reward model is typically learned from your preference data (the same data used in step 3, but you train a separate or parallel model that outputs a scalar reward).

Policy Optimization

Use a Proximal Policy Optimization (PPO) method or similar to adjust your SFT model in small steps so it maximizes the reward (as predicted by the reward model) while not deviating too far from the original model.

4.2 Code Flow for RLHF

A typical sequence using a library like Hugging Face TRL (Transformers Reinforcement Learning):

Train or Load a Reward Model

reward\_model.py

This might be a small feedforward model on top of a transformer that outputs a single scalar per token or per sequence.

PPO Trainer

from trl import PPOTrainer

# 1. Initialize your PPO trainer with the SFT model (policy) and the reward model

ppo\_trainer = PPOTrainer(

policy\_model=sft\_model,

reward\_model=reward\_model,

tokenizer=tokenizer,

\*\*ppo\_config

)

# 2. Prepare your training samples (prompts)

for batch in dataloader:

prompts = batch["prompt"]

# The model generates responses

responses = ppo\_trainer.generate(prompts, max\_new\_tokens=128)

# PPO step

train\_stats = ppo\_trainer.step(prompts, responses)

Update The policy in small steps, ensuring you track metrics like KL divergence (to prevent the model from drifting too far from the initial SFT checkpoint).

4.3 Additional Orchestrator Scripts

rlhf\_pipeline.py:

Load SFT model.

Load or train the reward model.

Run PPO with a set of prompts.

Save the new policy checkpoint.

preference\_pipeline.py (If you want a separate step to prepare or train the reward model from the preference data).

5. Reinforcement Learning from AI Feedback (RLAF)

RLAF is a variant of RLHF where the “human feedback” can be partially or fully replaced by AI annotations.

The pipeline is similar to RLHF, except your “preference data” might come from another “expert” or ensemble model. The structure is the same—reward model + policy optimization.

6. Putting It All Together

Start Simple: Implement or refine your SFT pipeline (train.py, infer.py, pipeline.py).

Add Preference Data (optional step) to produce a “preferred response” dataset or train a reward model.

Implement an RL pipeline:

Create a reward\_model.py to define and train your reward model.

Create PPO or other RL scripts to orchestrate policy optimization.

Integrate everything in your orchestrator:

preference\_pipeline.py for preference data loading + reward model training.

rlhf\_pipeline.py for the actual RL loop (like PPO).

7. Example RLHF Microservice Flow (Conceptual)

Below is a conceptual diagram of how your microservices could communicate if you really want them decoupled:

+----------------+ +--------------------+ +------------------+

| data\_service | (load) | model\_service | (train) | orchestrator |

| - data\_loader ----> | - reward\_model -----> | - RLHF pipeline |

| - preference\_data\_loader | - policy\_model | | |

| +--------------------+ +------------------+

| (optional) transform <---- feedback <----- +--- PPO steps

+----------------+

Data Service: loads both supervised data (for SFT) and preference data.

Model Service: contains the SFT model, the reward model, and the RL algorithms.

Orchestrator: calls the appropriate microservice functions in sequence (e.g., train reward model, run PPO, evaluate, etc.).

8. High-Level Steps to Implement Over Time

Finalize/Refine SFT

Standard fine-tuning on your domain data.

Possibly use parameter-efficient methods like LoRA or PEFT if model is large.

Gather Preference Data

E.g., (prompt, response A, response B, which is better?).

Store in data/preferences/.

Write preference\_data\_loader.py to parse and feed into the next step.

Train a Reward Model

reward\_model.py.

This is a classifier or regression model on top of a language model that outputs a scalar.

Use the preference data to learn a preference function.

RL Fine-Tuning (RLHF / RLAF)

Use your SFT model as the policy initialization.

Use the reward model in the PPO loop.

Save the updated model as model\_service/rlhf\_policy.py or similar.

Integration and Testing

Write test cases in tests/test\_rlhf.py to validate each step.

Check overall pipeline in rlhf\_pipeline.py.

Deployment (Optional)

If you want to run this at scale or in production, containerize each microservice and orchestrate with Docker Compose or Kubernetes.

Or you can keep it local if your training runs are smaller.

9. Final Thoughts

Incremental Approach: You don’t have to implement RLHF or preference fine-tuning all at once. Start with SFT, gather preference data, then build out the reward model and RL steps.

Libraries:

Hugging Face Transformers for model loading & tokenization.

Hugging Face TRL (or trlX) for PPO-based RLHF.

PEFT for parameter-efficient fine-tuning if the base model is large.

Project Modularity: Your microservices architecture helps keep the code structured: data loading in one place, model code in another, orchestrator scripts for each stage (SFT, preference, RL).

By following this blueprint, you’ll be able to start with standard supervised fine-tuning, then expand to preference fine-tuning, and eventually incorporate RLHF or RLAF as your project requirements evolve. This sets a strong foundation for advanced model alignment and customization in your domain.