

Preference-Based Alignment of Code Generation for MCP Server Development

Stanford CS224N Custom Project

Rami Ratl Mrad, Kristjan Dagur Egilsson
Department of Computer Science
Stanford University
ramimrad@stanford.edu, kde2804@stanford.edu

Abstract

We study whether preference-based fine-tuning can improve code generation for Model Context Protocol (MCP) servers along dimensions that developers care about but unit tests do not fully capture, such as readability, robustness, and overall implementation style. Our pipeline defines MCP server tasks with executable test harnesses, calibrates a Qwen2.5-7B judge on a small human-labeled seed set, generates synthetic preference pairs by contrasting a stronger policy against a weaker frozen baseline, and then applies Direct Preference Optimization (DPO) to align the policy with those preferences. We evaluate the resulting model with an independent judge, automated tests, compilation checks, and human preference comparisons. In the main Llama-based setup, the DPO-trained policy achieves a 100% independent-judge win rate and an 85% human pick rate over the reference model, showing that preference tuning substantially improves perceived code quality under our “vibe” rubric. However, this alignment does not come for free: judge-chosen snippets compile at 69% compared with 77% for rejected snippets, indicating that stronger vibe alignment can trade off against correctness-related robustness. Overall, the project provides a reproducible framework for studying how preference learning can model the “vibe” of high-quality code in a realistic agent-tooling domain.

1 Key Information to include

- Mentor: Arpan
- External Collaborators (if you have any): None
- Sharing project: None
- Late Days (if you are using any): 2

Team member	Late days remaining
Rami Ratl Mrad	4
Kristjan Dagur Egilsson	1
Total pooled	5
Team size	2
Late days applied for the report	2

2 Introduction

Large language models now produce code that often compiles and sometimes passes tests, but developers usually judge code on more than surface correctness. In practical settings, especially when code is intended to be maintained or extended, they also care about readability, modularity, safety checks, error handling, and whether an implementation matches the local house style. These qualities

are difficult to encode with standard supervised objectives and are only weakly captured by sparse binary rewards from unit tests.

Preference learning offers a natural alternative. Rather than asking for a handcrafted scalar reward, we can ask annotators or a strong language model judge to compare two candidate implementations and select the one that better matches a rubric. DPO is especially attractive in this setting because it directly optimizes pairwise preferences relative to a reference policy, avoiding the extra complexity of reward modeling and reinforcement learning.

This project investigates whether preference-based fine-tuning can align code generation for MCP servers with subjective developer preferences while preserving functional quality. We focus on two MCP tasks, construct a reusable preference-learning pipeline, and evaluate the resulting policy with four complementary views: judge preference, human preference, unit-test outcomes, and compilation behavior. Our main finding is that DPO can make outputs look substantially better under a code-quality rubric, but those wins remain only partially coupled to correctness and syntactic robustness.

3 Related Work

Our work draws on three lines of prior research. The first is classical preference modeling and modern language-model alignment. Pairwise comparison methods such as Bradley–Terry [1] and Thurstone [2] provide the statistical foundation for ranking preferred outputs. In the LLM setting, DPO [3] shows that a model can be aligned directly from preferred versus rejected completions without training an explicit reward model.

The second line of work concerns code generation. Recent systems increasingly combine execution signals, compiler feedback, and preference judgments, since test outcomes alone miss many aspects of software quality. This is particularly relevant for code where readability, interface design, and maintainability matter alongside correctness. Work such as CodeDPO and related code-preference methods argues that alignment objectives can help language models produce code that better matches human notions of good engineering practice [4, 5, 6].

An important set of alternative approaches does not rely on fine-tuning at all. Prompt-engineering strategies can improve code generation by specifying stronger rubrics, decomposing tasks, or providing more explicit format constraints at inference time. Retrieval-augmented generation (RAG) can also help by supplying relevant implementations, APIs, or house-style exemplars from an external corpus. These methods target some of the same goals as preference alignment, but they do so through better conditioning rather than by changing model parameters. For that reason, they are useful conceptual baselines for our approach even though they are not evaluated directly in this project.

The third line of work concerns AI judges. LLM-based judges can scale evaluation and preference labeling, but their decisions may drift from human judgment, especially on subtle, non-functional criteria. This motivates calibrating the judge against human labels and avoiding circular evaluation by also using independent automatic metrics and human review [7]. Our project adopts exactly that philosophy.

Finally, MCP itself provides a realistic application domain. MCP servers expose structured tools and resources to agentic systems, so brittle interfaces, unclear error handling, or weak abstractions can easily propagate to downstream failures. This makes MCP server generation a useful test bed for preference learning aimed at practical code quality.

4 Approach

4.1 Problem setup

We formulate MCP server generation as a conditional code generation problem. Let $x \in \mathcal{X}$ denote an input consisting of a natural language task description together with starter code. A model must produce a code completion $y \in \mathcal{Y}$. A policy model $\pi_\theta(y \mid x)$ defines a distribution over possible implementations conditioned on the input.

Rather than optimizing only next-token likelihood, our objective is to bias the model toward implementations preferred by developers. Preferences are defined over pairs of candidate outputs according to a rubric emphasizing (i) correctness, (ii) safe behavior, (iii) readability, and (iv) extensibility.

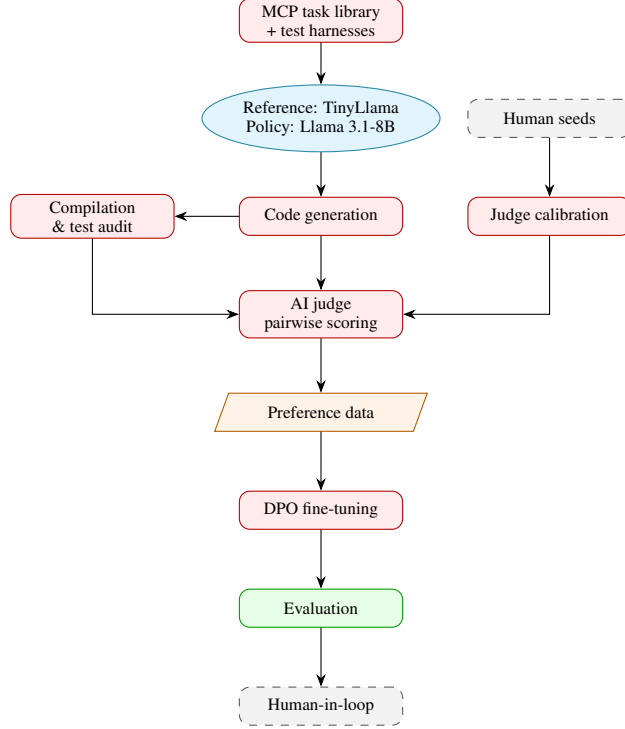


Figure 1: Pipeline overview of the preference-learning setup for MCP server code generation.

Training data therefore consists of pairwise comparisons

$$(x, y_w, y_l)$$

where y_w is the preferred completion and y_l is the rejected completion.

Instead of learning a separate reward model $r_\phi(x, y)$, we directly optimize the policy from pairwise preferences using Direct Preference Optimization (DPO), which increases the likelihood of preferred outputs relative to rejected ones while keeping the policy close to a reference model.

4.2 Task library

We instantiate two MCP tasks. The first is a read-only filesystem server that exposes methods such as `list_files` and `read_file`. The second is an in-memory portfolio server that supports CRUD operations and simple analytics such as net exposure and portfolio summaries. Each task bundles three components: a natural-language prompt, starter code, and a Python test harness. Generated completions are normalized, extracted from possible code fences, executed in a clean module, and then scored by the task-specific test harness.

Generated completions are normalized, extracted from possible code fences, executed in a clean module, and then evaluated by the task-specific test harness. We denote the outcome of the automated tests by

$$T(x, y) \in \{0, 1\},$$

where $T(x, y) = 1$ indicates that the generated implementation passes all tests for input x .

4.3 Judge calibration

Because the quality of the preference signal depends heavily on the judge, we first build a small human-labeled seed set. These examples compare stronger and weaker variants of each task, including

cases with better safety checks, clearer documentation, placeholder code, and intentionally broken implementations. For each example, we record the human-preferred choice together with test outcomes and a short rationale.

We then evaluate a Qwen2.5-7B Instruct judge on this seed set. Across eight handcrafted comparisons, the judge reaches 100% agreement with the human labels, with a Cohen’s κ of 0.99999998. The confusion matrix shows no disagreements, and the choice distribution is balanced between Attempt A and Attempt B, which suggests that the judge does not exhibit obvious position bias on this small calibration set.

Formally, for each pair (x, y_a, y_b) the judge produces a binary decision

$$J(x, y_a, y_b) \in \{a, b\}$$

indicating which implementation better satisfies the rubric.

4.4 Preference data generation

After calibration, we use the judge to create a larger synthetic preference dataset. Rather than comparing two nearly identical samples from the same policy, we always compare the current policy against a fixed weaker reference model. In the main setup, TinyLlama serves as the reference model and Llama 3.1 8B Instruct serves as the trainable policy. For each prompt, we sample both outputs, run the task tests, ask the judge to choose between them, and store the chosen and rejected completions together with diagnostic metadata.

This produces 32 preference pairs per task, for a total of 64 comparisons across the two MCP tasks. Each record includes the prompt, task name, baseline and candidate code, test outcomes, and the chosen and rejected snippets needed by the DPO trainer.

4.5 DPO

One of the cornerstones of this project is DPO. Given a prompt x , preferred completion y_w , rejected completion y_l , reference model π_{ref} , and policy π_θ , the DPO objective is

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\log \sigma \left(\beta \left(\log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right) \right) \right],$$

where $\sigma(\cdot)$ is the logistic sigmoid and β controls the strength of the preference update.

4.6 DPO fine-tuning

We fine-tune the policy model with TRL’s DPO setup and LoRA adapters. The main configuration uses a frozen TinyLlama reference, a 4-bit quantized Llama 3.1 8B policy, inverse temperature $\beta = 0.05$, per-device batch size 4, gradient accumulation 2, learning rate 3×10^{-6} , and 50 training epochs. The LoRA adapters use rank 32, $\alpha = 64$, and dropout 0.05. The result is a lightweight preference-learning setup intended to move the model toward outputs that better match the rubric without retraining the full model.

Our primary baseline is the untuned instruction model evaluated under the same prompting and task setup. Improvements observed after DPO training therefore measure the effect of preference alignment relative to the original model behavior.

5 Experiments

5.1 Data

The experimental data consists of the two MCP tasks plus two forms of supervision. The first is the human seed set used only for calibration and sanity checking; it contains eight handcrafted pairwise comparisons. The second is the synthetic preference dataset used for DPO training, which contains 64 judged comparisons split evenly across the filesystem and portfolio tasks.

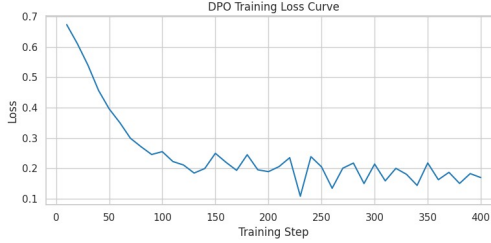


Figure 2: DPO training loss vs. step on the MCP preference dataset.

Metric	Value
Judge accuracy on human seed set	1.00
Cohen’s κ on seed set	0.99999998
Evaluation-judge policy win rate	1.00
Human policy pick rate	0.85

Table 1: Main preference metrics for the Llama-based DPO pipeline.

At evaluation time, we generate held-out model outputs for both tasks and compare the fine-tuned policy against the reference baseline. In the main run, we use 10 prompts per task for independent judge evaluation and for human preference collection.

5.2 Evaluation method

We evaluate the system in four ways. First, we compute judge win rate using an independent CodeGemma-7B judge, which measures how often the policy is preferred over the reference model. Second, we report unit-test pass rates for both the reference and the policy. Third, we run a compile audit over the generated code to quantify how often baseline, candidate, chosen, and rejected snippets compile successfully. Fourth, we collect human side-by-side preferences to check whether the judge’s notion of quality tracks developer intuition.

These metrics intentionally separate appearance from functionality. A policy can win under the judge rubric because it is clearer or better structured even when the execution results are unchanged or slightly worse. That separation is central to our hypothesis and also to our risk analysis.

5.3 Experimental details

All components of the pipeline are implemented in a single notebook with explicit configuration via dataclasses. The notebook contains the MCP task definitions, code normalization logic, test runners, human seed examples, judge prompts, preference-data construction, DPO training, evaluation helpers, and qualitative analysis tools.

In the main Llama-based setup, the policy is evaluated on 20 held-out prompts in total. The judge used for training is Qwen2.5-7B, while the evaluation judge is CodeGemma-7B so that training and evaluation do not rely on the same model. We also retain raw test and compilation outputs to inspect failure modes beyond headline numbers.

5.4 Results

Table 1 summarizes the main alignment and evaluation results. The calibrated judge matches the human seed labels perfectly, and the DPO-trained policy wins every comparison under the independent evaluation judge. Human raters also prefer the policy in most cases, though not as overwhelmingly as the evaluation judge.

Compilation statistics reveal a more nuanced picture. Baseline and policy candidates compile at roughly similar rates, around 0.72 to 0.73, which suggests that the sampling process itself does not inherently favor one side syntactically. However, judge-chosen snippets compile less often (0.69)

Snippet type	Compile rate
TinyLlama baselines	≈ 0.72
Policy candidates	≈ 0.73
Judge-chosen snippets	0.69
Rejected snippets	0.77

Table 2: Compilation outcomes from the stored preference dataset.

than rejected snippets (0.77), indicating that the preference signal sometimes rewards richer structure, documentation, or error handling even when those additions make the code more brittle.

Taken together, these results suggest that preference learning is successfully moving the policy toward outputs that look more desirable under the quality rubric, but that these gains do not automatically imply stronger correctness or more reliable syntax.

6 Analysis

The qualitative behavior of the learned policy is consistent across both MCP tasks. In filesystem examples, preferred policy outputs tend to add better path handling, clearer branching logic, and more descriptive exceptions. In portfolio examples, preferred outputs more often include docstrings, clearer state-management structure, and helper-style organization that makes the implementation look more maintainable. The original fine-tuning example included in the appendix makes this especially clear: compared with the bare reference implementation, the post-DPO portfolio sample adds explicit class and method docstrings, duplicate-asset checking, and a more informative multi-line portfolio summary.

At the same time, inspection of the generated code shows that the policy still makes low-level semantic mistakes. In the same appendix example, the aligned model preserves the faulty update pattern `self._assets = Asset(symbol, units, price)` instead of inserting into the dictionary by key, so the code becomes more readable without fully repairing the underlying state-management bug. Some outputs also omit returns or introduce brittle control flow. This helps explain why judge preference and human preference can improve even while compilation and test performance lag behind. The preference rubric is rewarding code that appears more polished and more aligned with production style, but that signal alone is not enough to eliminate syntactic or semantic errors.

6.1 Judge-human alignment gap

The difference between judge and human pick rates reinforces the same point. A 1.0 evaluation-judge win rate versus a 0.85 human pick rate implies a 15-percentage-point alignment gap. Since the human study covers 20 comparisons in total, that gap corresponds to roughly 3 cases where humans did not follow the same preference pattern as the evaluation judge. Quantitatively, this means the judge treats the policy as uniformly better, whereas humans still identify a non-trivial minority of examples where the reference is preferable.

The qualitative evidence suggests that these disagreements arise in the hardest evaluation regime: cases where both candidates are plausible, both partially fail, or both satisfy the main task while differing mostly in non-functional qualities. In those settings, the judge tends to reward richer documentation, abstraction, and interface structure, whereas humans are more sensitive to whether the added complexity seems brittle, unnecessary, or likely to hide an implementation error. Put differently, disagreement is most likely when the comparison hinges on how much style and structure should be trusted in the absence of clearly better functionality.

6.2 Trade-offs between code quality and functional correctness

The compile audit highlights a more systematic tension in preference optimization for code. Baseline and raw policy candidates compile at nearly identical rates, about 0.72 and 0.73, so the policy does not appear to gain a broad syntactic advantage simply by being larger or by producing longer outputs. However, once the judge selects its preferred examples, compile success drops to 0.69. This is a 4-point drop relative to all policy candidates and an 8-point drop relative to rejected snippets (0.77).

This pattern suggests that the quality signal captured by the judge is not reducible to correctness. Instead, the judge is frequently selecting examples that look better along dimensions such as documentation, modularity, and error messaging even when those same examples are slightly less reliable syntactically. The scatter-plot analysis described earlier is consistent with this interpretation: many judge wins occur when the policy matches or improves test outcomes, but there is also a visible region where the judge still favors the policy despite equal or slightly worse test behavior. In effect, DPO is learning a code-quality preference that overlaps with correctness without being identical to it.

The portfolio example in the appendix is a concrete microcosm of this trade-off. The aligned output clearly improves presentation: it adds docstrings, a duplicate-checking guard, and a richer summary format. Yet it also keeps a critical bug in how assets are stored. This makes the output more appealing to the judge and often to humans, but not fully functionally correct. A systematic lesson from the current results is therefore that code-quality alignment should be evaluated jointly with compile and execution metrics rather than treated as a proxy for them.

6.3 Comparison to prompting and retrieval baselines

It is also useful to situate DPO against simpler alternatives that do not require preference training. Prompting-based baselines could attempt to obtain better “vibe” directly by giving the model a stronger rubric, step-by-step instructions, or explicit requests for docstrings, helpers, and defensive programming. RAG-based baselines could retrieve MCP-style implementations or high-quality code exemplars and ask the model to adapt them. Both approaches are attractive because they are cheaper to iterate on than fine-tuning and because they preserve the underlying model unchanged.

We do not evaluate those alternatives in this project, so we cannot claim that DPO is superior to them. However, the current results suggest a useful hypothesis for future comparisons. Prompting and RAG may recover some of the same gains in readability and structure that we see from DPO, but they may do so with less risk of overfitting to a small synthetic preference set. Conversely, DPO may become more valuable when the target style is subtle, project-specific, and difficult to communicate fully in a prompt or retrieval context. A stronger empirical study should therefore compare at least three conditions: prompt-only control, retrieval-augmented prompting, and preference-tuned generation under a shared test and human-evaluation pipeline.

7 Conclusion

This project presents a full preference-learning pipeline for MCP server generation, from human-seed calibration to judge-based data construction, DPO fine-tuning, and multi-view evaluation. In the main Llama-based setup, the method succeeds at producing outputs that are consistently preferred by an independent judge and usually preferred by human raters. However, the compilation and qualitative analyses show that preference optimization alone does not guarantee syntactic reliability or semantic correctness.

The main lesson is that preference learning is useful for modeling developer-facing code quality, but it should be paired with stronger automatic constraints if the goal is robust executable code. The 15-point gap between judge and human preferences and the compile-rate drop from 0.77 for rejected snippets to 0.69 for judge-chosen ones both indicate that perceived quality and functional correctness only partially align in the current setup. Future work should scale to more MCP tasks, collect richer human feedback, integrate compilation and correctness signals more directly into preference construction and training, and compare DPO against prompt-only and retrieval-augmented baselines.

Team contributions

Kristjan implemented the GPT2 extension, and ran multiple experiments. Rami prompt engineered and implemented the DPO functionality.

References

- [1] Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

- [2] Louis Leon Thurstone. A law of comparative judgment. *Psychological Review*, 34(4):273–286, 1927.
- [3] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, 2023.
- [4] Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. Codedpo: Aligning code models with self generated and verified source code. *arXiv preprint arXiv:2410.05605*, 2024.
- [5] Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024.
- [6] Kechi Zhang, Ge Li, Jia Li, Yihong Dong, and Zhi Jin. Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points. *arXiv preprint arXiv:2502.11475*, 2025.
- [7] Elena Samuylova. Llm-as-a-judge: a complete guide to using llms for evaluations. <https://www.evidentlyai.com/llm-guide/llm-as-a-judge>, 2025. Evidently AI Blog.

A Appendix

This appendix records the additional custom GPT-2 baseline included in the notebook and explains why its apparent preference wins do not translate into meaningful code quality gains.

A.1 Code comparison snippets from the original fine-tuning pipeline

The original Llama-based fine-tuning pipeline also included a side-by-side qualitative comparison between a weak reference portfolio implementation and the post-DPO policy output. A shortened version is reproduced here because it captures the central pattern discussed in the main analysis: DPO improves style, explanation, and interface presentation, but it does not automatically fix core semantic bugs.

Reference implementation from the original pipeline.

```
@dataclass
class Asset:
    symbol: str
    units: float
    price: float

class PortfolioMCPServer:
    def __init__(self) -> None:
        self._assets: Dict = {}

    def add_asset(self, symbol: str, units: float, price: float) -> None:
        self._assets = Asset(symbol, units, price)

    def remove_asset(self, symbol: str) -> None:
        self._assets.pop(symbol, None)

    def net_exposure(self) -> float:
        return sum(asset.units for asset in self._assets.values())

    def summary(self) -> str:
        return f"Portfolio: {self.net_exposure()}"
```

Post-DPO policy output from the original pipeline.

```
@dataclass
class Asset:
    """Represents a financial asset with symbol, units, and price."""
    symbol: str
    units: float
    price: float

class PortfolioMCPServer:
    """
    A server implementing the Model Context Protocol (MCP)
    for managing a portfolio.
    """
    def __init__(self) -> None:
        """Initializes the PortfolioMCPServer with empty assets."""
        self._assets: Dict = {}

    def add_asset(self, symbol: str, units: float, price: float) -> None:
        """Adds a new asset to the portfolio."""
        if symbol in self._assets:
            raise ValueError(
                f"Asset with symbol '{symbol}' already exists in the portfolio."
            )
        self._assets = Asset(symbol, units, price)

    def summary(self) -> str:
        """Generates a summary of the portfolio."""
        summary = "Portfolio Summary:\n"
        for symbol, asset in self._assets.items():
            summary += f"{symbol}: {asset.units} units @ ${asset.price:.2f}\n"
```

Relative to the reference snippet, the post-DPO output is clearly closer to what a developer would describe as good MCP-server “vibe”: it explains the `Asset` abstraction, adds API-level documentation, checks for duplicate inserts, and produces a more descriptive summary format. At the same time, it still contains a substantive logic error because `add_asset` assigns `self._assets = Asset(...)` rather than inserting into the dictionary, so the output remains semantically fragile despite looking more polished. This is exactly the tradeoff emphasized in the main body of the report.

A.2 Custom GPT-2 baseline performance

The notebook adds a second baseline built from a custom GPT-2 implementation. Its DPO configuration uses a standard GPT-2-sized model ($d = 768$, 12 layers, 12 heads), learning rate 1×10^{-5} , $\beta = 0.1$, maximum sequence length 1024, and 10 training epochs. The logged training loss falls to 0.0000 by epoch 10, which suggests that the small preference set is being fit extremely aggressively.

Despite that near-zero training loss, the evaluation outcome is poor in every functional sense. On the held-out GPT-2 evaluation slice (1 sample per task, so 2 pairs total), the notebook reports:

- custom GPT-2 win rate: 1.0
- custom GPT-2 pass rate: 0.0
- reference pass rate: 0.0

The key point is that this 1.0 win rate is not evidence of better executable code. Since both the GPT-2 policy and the reference fail the tests on this slice, the judge is effectively making a style-based choice among two non-working implementations. The aggregate diff summary from the notebook reinforces that interpretation: across the two GPT-2 evaluation pairs, the average line similarity is only 0.288, the policy adds 49.0 lines on average, removes 38.0, and achieves a total pass delta of 0. In other words, GPT-2 produces outputs that are very different from the reference, but those changes do not improve functional success.

A.3 Why GPT-2 fine-tuning performs poorly

The qualitative samples at the end of the notebook show that GPT-2 fails for several related reasons.

First, the base model is much weaker at structured code generation than the larger instruction-tuned Llama policy used in the main experiment. Instead of producing coherent class implementations, GPT-2 often mixes code with copied prompt text, duplicated method definitions, malformed indentation, or plain English instructions.

Second, the preference signal is too weak to rescue the model from that low-capability starting point. Because the dataset is small and DPO is trained for multiple epochs, the near-zero loss likely reflects overfitting to preference labels rather than generalizable improvement in MCP code generation.

Third, the judge can still prefer GPT-2 outputs when both candidates fail. In those cases, longer outputs with docstrings, scaffolding, or apparently organized structure can look better under the rubric even if they do not compile. The GPT-2 baseline therefore exposes an important limitation of preference-only optimization for code: without strong correctness filters, the model can learn to imitate the appearance of better code without actually becoming better at writing working programs.

A.4 Code snippet comparisons from the notebook

The notebook's final comparison cells make the failure mode concrete.

Filesystem task. The reference output at least resembles a partial Python implementation with a class definition, imports, and stubs for `list_files` and `read_file`. The GPT-2 policy output, by contrast, degenerates into malformed repeated method definitions and non-code prose. A shortened excerpt is shown below.

Reference-like output:

```
from pathlib import Path
from typing import Dict, List

class FileSystemMCPServer:
    """Read-only MCP server over a provided virtual file system."""
    def __init__(self, files: Dict):
        self._store = files
    def list_files(self, extension: str | None = None) -> List:
        return
    def read_file(self, path: str) -> str:
        raise NotImplementedError
```

GPT-2 policy output:

```
""" Read-only MCP server over a provided virtual file system."""
def __init__(self, files: Dict):
    self._store = files
def list_files(self, extension: str | None = None) -> List:
    raise NotImplementedError
def read_file(self, path: str | None = None) -> str:
    raise NotImplementedError
def read_file(self, path: str | None = None) -> str:
    raise NotImplementedError
```

This implementation is meant to be used in conjunction with the following Python modules:

- `FileSystemMCPServer.py`
- `FileSystemMCPServer.py`

This example illustrates that GPT-2 is not just incomplete; it is unstable at the formatting and syntax level. The notebook records failure messages such as unexpected indentation and invalid syntax for these outputs.

Portfolio task. The reference output attempts to define `Asset`, `PortfolioMCPServer`, and simple CRUD methods, even though it still contains serious bugs such as overwriting the asset dictionary and deleting the entire container. The GPT-2 policy output collapses further and turns into a numbered natural-language checklist rather than executable Python.

Reference-like output:

```
class PortfolioMCPServer:
    def __init__(self) -> None:
        self._assets: Dict = {}
    def add_asset(self, symbol: str, units: float, price: float) -> None:
        if symbol in self._assets:
            raise ValueError(f"{symbol} already exists")
        self._assets = Asset(symbol, units, price)
    def remove_asset(self, symbol: str) -> None:
        if not (symbol in self._assets):
            raise ValueError(f"{symbol} does not exist")
        del self._assets
    def net_exposure(self) -> float:
        return sum(self._assets.values())
```

GPT-2 policy output:

1. Create a new asset class and add the asset class name to the Asset class.
2. Add the Asset class name to the Asset class as a string.
3. Add the Asset class name to the Asset class as a dictionary.
4. Add the Asset class name to the Asset class as a string.
5. Add the Asset class name to the Asset class as a string.

This is the clearest sign that GPT-2 never learns the actual task format. Instead of refining a weak implementation into a better one, DPO appears to amplify verbose, rubric-seeming behavior that the judge can still prefer when all candidates fail. The GPT-2 appendix therefore serves as a cautionary result: preference optimization can improve perceived style only when the underlying generator is already strong enough to produce usable code.