HDPO: ALIGNING LARGE LANGUAGE MODELS TO SOFTWARE ARCHITECTURE VIA HIERARCHICAL PREFERENCE OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models struggle with **software evolution** in complex repositories, often taking a "Semantic Shortcut" that ignores architectural hierarchy. We introduce **Hierarchical Direct Preference Optimization (HDPO)**, a novel alignment framework that enforces structural consistency by decomposing feature generation into a rigorous three-tier mapping of *Modules*, *Tasks*, and *Files*. Across major open-source ecosystems (e.g., Django, Kafka, Kubernetes), HDPO achieves an average structural accuracy of 40.3%, drastically outperforming Step-SFT (14.7%), Step-Weighted-SFT (16.5%), and Step-DPO (19.0%). Similarly, regarding Average Weighted Impact across these ecosystems, HDPO (51.0%) decisively outperforms Step-DPO (25.0%), Step-Weighted-SFT (22.3%), and Step-SFT (20.7%).

1 INTRODUCTION

In large-scale software ecosystems (e.g., search engines, ad platforms Tang et al. (2015); Kohavi et al. (2013)), a fundamental dichotomy exists between module code quality and system-wide feature velocity Ossei-Owusu (2017). Historically, consumer-facing tech companies have logically siloed developers into specialized teams (e.g., Client, Data, Infrastructure) to manage domain complexity Dragoni et al. (2016). Because engineers focus deeply on their own modules, localized syntax synthesis, passing unit tests (e.g., pass@k), and clearing CI/CD pipelines are rarely the primary development blockers Du et al. (2024). However, a critical breakdown occurs during staged experimentation when enforcing a multidimensional launch matrix Xie & Aurisset (2016). This friction arises from the exponentially complex concurrent orchestration of disjoint modules (e.g., ML models, serving infrastructure, and UIs) Sculley et al. (2015).

To mitigate this friction, organizations establish dedicated Engineering Productivity (EngProd) teams to construct **localized integration testing environments** Arguelles (2023); Jaspan & Green (2025). Yet, these bespoke environments typically cover only **a few modules** and require multiple quarters of engineering effort to build, burdening developers with complex configurations merely to triage system failures Waseem et al. (2026). Even with EngProd support, guaranteeing implementation correctness during the experimental stage remains difficult, dragging developers into reactive "debug-on-matrix-fail" loops Amershi et al. (2019); Breck et al. (2017)(detail see Appendix B). For instance, Figure 1.1 shows that in normalized production logs from a large-scale commercial advertising platform, code remediation at the 1% rollout stage (P50: 4d) takes **four times longer** than the experimentation window itself (P50: 1d). This delay occurs because teams operate blindly, discovering missing requirements only through launch matrix discrepancies **without comprehensive testing**.

In other words, if developers know exactly **what needs to be implemented** within their own modules to launch a feature, they can seamlessly write the code—or employ LLMs to generate it—and easily pass local unit(pass@k) and CI/CD tests. To resolve the architectural bottleneck, HDPO leverages the final historical "chain-of-patches" as the **ground-truth task scope** for an end-to-end feature across all required modules. HDPO employs LLMs to decompose milestone structures upfront by: (1) mapping features to modules, (2) deriving specific implementation tasks, and (3) predicting affected file paths. As illustrated by the subtle dependencies mandating strict execution orders in Figure 1(3), this hierarchical data forms a Directed Acyclic Graph (DAG) of patch metadata (Figure 1(4)). By generating a complete blueprint of required tasks across a concurrent repository,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

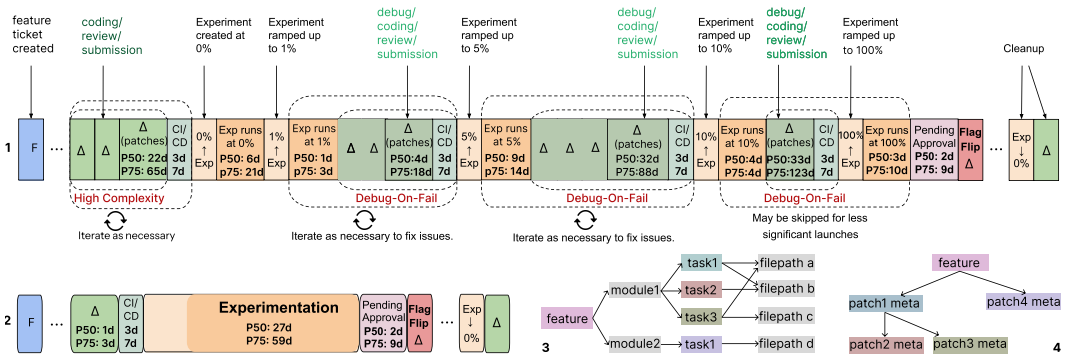


Figure 1: **The Structural Transformation of Milestone Delivery. (1) normalized logs from the advertising platform :** Feature delivery is fragmented. *Development Phase:* The initial coding/review cycle is slow (P50: 22d, P75: 65d) due to sequential dependency management. Note: P50 denotes the Median duration, and P75 denotes the 75th Percentile. *Experiment Phase:* Rollout is interrupted by reactive "debug/coding/review" loops. For example, stabilizing the 10% ramp alone takes substantial time (P50: 32d, P75: 178d). **(2) HDPO Workflow:** Development is "shifted left." *Development Phase:* Parallel coding reduces the initial cycle to atomic units (P50: 1d, P75: 3d). *Experiment Phase:* Because architectural constraints are resolved upfront, the experimentation phase becomes a consolidated, purely observational period (P50: 27d, P75: 59d) without code churn. **(3) Hierarchical Context:** Tasks sharing a parent module (e.g., *task1* and *task2* under *module1*) inherit the same directory path. **(4) Dependency Graph:** These couplings necessitate a directed acyclic graph where nodes are "Patch Metadata" (descriptions and filepaths) rather than isolated file changes.

HDPO eliminates late-stage debug-on-fail loops. It provides a **skeleton** for parallelized "shift-left" development (Figure 1(2)), allowing engineers to leverage highly efficient localized coding practices within their isolated domains.

We provide a comprehensive empirical evaluation across four diverse domains spanning both proprietary (Ads) and open-source (Kafka, Kubernetes, Django) ecosystems. The results demonstrate that Across major open-source ecosystems (e.g., Django, Kafka, Kubernetes), HDPO achieves an average structural accuracy of 40.3%, drastically outperforming Step-SFT (14.7%), Step-Weighted-SFT (16.5%), and Step-DPO (19.0%). Finally, to quantify practical deployment impact, we evaluate using **Weighted Impact (WI)** Castillo et al. (2024)—an **industry-standard** metric utilized by EngProd teams in the matrix driven organization(e.g. Ads) that treats end-to-end feature velocity as a reliable proxy for structural complexity. Measured by WI, HDPO (51.0%) decisively outperforms Step-DPO (25.0%), Step-Weighted-SFT (22.3%), and Step-SFT (20.7%)., proving its ability to resolve the high-complexity, time-consuming integration scopes that standard models fail to address.

2 Related Work

Preference Optimization Algorithm While Supervised Fine-Tuning (SFT) instills domain knowledge Ouyang et al. (2022), it often overlooks subtle reasoning errors in complex planning. Alignment traditionally relied on computationally expensive **PPO** Schulman et al. (2017). Recent efficient alternatives include **DPO** Rafailov et al. (2023a) (evaluated herein as **Step-DPO** Lai et al. (2024) for step-by-step reasoning beyond context limits), **ORPO** Hong et al. (2024), **SimPO** Meng et al. (2024), and critic-free **GRPO** Shao et al. (2024). Furthermore, **Token-DPO** Zeng et al. (2024) optimizes micro-token generation. However, these methods rarely address the macro-architectural **hierarchical planning correctness** required for system-wide feature velocity (Weighted Impact) across concurrent modules. To directly solve this architectural bottleneck, our methodology operates as a Step-Weighted-DPO—termed **Hierarchy-DPO (HDPO)**—generating upfront, system-wide task blueprints.

Reasoning Priors and Agentic Planning Beyond architectural modifications, inference-time strategies like **Chain-of-Thought (CoT)** Wei et al. (2022) and **Tree of Thoughts (ToT)** Yao et al. (2023) attempt to improve logic via intermediate steps, while **ReAct** Yao et al. (2022) and **Reflexion** Shinn

et al. (2023) interleave reasoning with interaction. However, these **prompt-driven heuristics** face severe scalability limits in large-scale software evolution: CoT suffers from **error accumulation** Turpin et al. (2023), and ReAct agents often devolve into **myopic loops** that violate global constraints Jimenez et al. (2023). HDPO diverges by *internalizing* this hierarchical planning structure directly into the policy parameters. This shifts the burden of correctness from **fragile prompting heuristics** to a native behavioral prior, ensuring robustness where purely inference-based methods fail.

LLMs for Code Generation and Software Evolution While models like Codex Chen et al. (2021), StarCoder Li et al. (2023), and DeepSeek-Coder-V2 DeepSeek-AI et al. (2024) excel at generating isolated functions from scratch, agents like OpenHands Wang et al. (2024) and MetaGPT Hong et al. (2023) struggle with precise modification of large-scale, legacy codebases Jimenez et al. (2023). To address this, three strategies have emerged, though each faces limitations. **Retrieval-Augmented Generation (RAG)** (e.g., RepoBench Liu et al. (2023b), GraphRAG Edge et al. (2024)) often relies on semantic similarity, missing deep structural dependencies. **Long-Context** models degrade significantly beyond 32k tokens Li et al. (2025), suffering from the "lost-in-the-middle" phenomenon Liu et al. (2023a). Finally, **Agentic Workflows** (e.g., SWE-agent Yang et al. (2024), AutoCodeRover Zhang et al. (2024b)) and RL-optimized systems like **SWE-RL** Wei et al. (2025) excel at isolated issue resolution (optimizing for pass@k); however, they lack global architectural planning, often devolving into reactive debugging loops instead of system-wide feature orchestration.

3 Methodology: Hierarchical Direct Preference Optimization (HDPO)

3.1 Theoretical Formulation

Standard Direct Preference Optimization (DPO) Rafailov et al. (2023b) implicitly assumes that the utility of a generated sequence is uniformly distributed across all tokens. In rigid engineering tasks, however, this assumption fails: a single error in a high-level dependency renders the entire solution invalid, regardless of implementation fluency.

To correct this, we introduce a **Structure-Aware Implicit Reward**, denoted as $\hat{r}_\theta(x, y)$. Unlike standard DPO, we define this reward as the strictly importance-weighted summation of the log-probability ratios between the policy π_θ and the reference π_{ref}:

$$\hat{r}_\theta(x, y; \mathbf{W}) = \beta \sum_{t=1}^{T} \alpha_t \cdot \Bigg(\log \pi_\theta(y_t | x, y_{<t}) - \log \pi_{\text{ref}}(y_t | x, y_{<t}) \Bigg) \tag{1}$$

where β is the KL-divergence penalty coefficient and $\alpha_t \in \mathbf{W}$ is the structural weight derived from the semantic plane of token y_t.

By utilizing this weighted reward, the HDPO objective function $\mathcal{L}_{\text{HDPO}}$ is formalized as the expected logistic margin between the chosen trajectory y_w and the rejected trajectory y_l:

$$\mathcal{L}_{\text{HDPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \Bigg[\log \sigma \Bigg(\hat{r}_\theta(x, y_w; \mathbf{W}) - \hat{r}_\theta(x, y_l; \mathbf{W}) \Bigg) \Bigg] \tag{2}$$

The execution of HDPO is formalized in Algorithm 1. Unlike standard RLHF pipelines requiring separate reward models Christiano et al. (2017), HDPO operates as a single-stage supervised loop. The coefficient α_t in α maps directly to three layers:

- **Module Layer (Layer 1: Feature $\rightarrow$ Modules, $\lambda_{\text{strat}} \approx 4.0$):** Governs high-level decomposition where errors represent fundamental misunderstandings that render all downstream tasks invalid, necessitating a maximal penalty to enforce architectural alignment.

Algorithm 1 Hierarchical Direct Preference Optimization (HDPO)

Require: Dataset $\mathcal{D}$, Policy π_θ, Ref π_{ref}
Require: Layer Weights $\mathbf{W} = \{\lambda_{\text{strat}} = 4.0, \lambda_{\text{ctrl}} = 2.0, \lambda_{\text{impl}} = 1.0\}$
// 1. Curriculum Initialization
Sort $\mathcal{D}$ descending by structural complexity $\sum \mathbf{W}$.
Initialize sampler $\mathcal{S}$ over sorted $\mathcal{D}$.
for all minibatch $B \in \mathcal{S}$ **do**
  Extract inputs (x, y_w, y_l) and pre-computed structural map.
  // 2. Hierarchical Importance Assignment
  Construct weight vector $\boldsymbol{\alpha}$ by mapping layers to tokens:
    $\alpha_t \leftarrow \lambda_{\text{strat}}$ **if** $y_t \in$ Modules Layer
    $\alpha_t \leftarrow \lambda_{\text{ctrl}}$ **if** $y_t \in$ Task Layer
    $\alpha_t \leftarrow \lambda_{\text{impl}}$ **if** $y_t \in$ File Layer
  // 3. Forward Pass & Weighted Accumulation
  Compute logits for $\pi_\theta, \pi_{\text{ref}}$ on $[x; y_w]$ and $[x; y_l]$.
  For each trajectory $y \in \{y_w, y_l\}$:
    $\log \pi(y|x) \leftarrow \sum_{t=1}^{T} \alpha_t \cdot \log P(y_t|x, y_{<t})$
  // 4. Compute Structure-Aware Rewards (Eq. 1)
  $\hat{r}_\theta(y) \leftarrow \beta\big(\log \pi_\theta(y|x) - \log \pi_{\text{ref}}(y|x)\big)$
  // 5. Optimization Step (Eq. 2)
  $\mathcal{L}_{\text{HDPO}} \leftarrow -\log \sigma\big(\hat{r}_\theta(y_w) - \hat{r}_\theta(y_l)\big)$
  $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_{\text{HDPO}}$
end for

- **Task Layer (Layer 2: Module $\rightarrow$ Tasks, $\lambda_{\text{ctrl}} \approx 2.0$):** Defines the operational "skeleton" and dependency chains, receiving a significant weighting boost to prevent incomplete logic or broken execution graphs.
- **File Layer (Layer 3: Task $\rightarrow$ Files, $\lambda_{\text{impl}} = 1.0$):** Handles granular scoping and narrative. These tokens remain at unity weighting to preserve natural language fluency without overriding higher-order structural constraints.

By explicitly encoding specific weights, HDPO mathematically enforces *top-down* consistency: the cost function renders architectural hallucinations prohibitively expensive compared to minor scoping variances.

4 Data Construction: The Hierarchical Evolution Protocol

Training a model to respect architectural constraints requires a dataset that explicitly exposes the causal chain from high-level features to atomic file patches. We introduce the **Hierarchical Evolution Dataset (HED)**, constructed by reverse-engineering large-scale software repositories to recover lost traceability links.

4.1 Data Sources and Hierarchy Extraction

We augment our high-precision proprietary corpus (Ads Domain) with open-source repositories (e.g., Django, Kafka) by mining merged pull requests. To ensure reproducibility across domains, we normalize these datasets into a hierarchical schema mapping $\{pull_request \rightarrow feature, commit \rightarrow task, diff \rightarrow file_paths\}$. The proprietary corpus consists of anonymized historical development cycles from a large-scale industrial advertising system, where "ground truth" is derived from successfully merged and verified production changes.

Dynamic Complexity-Adaptive Decomposition. To standardize module granularity without relying on language-specific patterns, we model the repository as a hierarchical file system tree, where directories represent internal nodes and source files represent leaf nodes Devanbu (2012).

We utilize the Lowest Common Ancestor (d_{LCA}) to determine the module scope. However, this creates a dilemma: if the d_{LCA} (e.g., root) is taken as the module, the massive *file count* overwhelms

the context window Liu et al. (2023a); conversely, if defined strictly at the leaf-parent level, the sheer *number of distinct modules* saturates the prompt capacity. We resolve this via a threshold of $N_{max} = 60$ files. Empirical analysis confirms only rare dense outliers ($< 3\%$) requiring retrieval ranking.

Guided by this threshold, we map a commit C to a set of modules $\mathcal{M} = \{m_1, \ldots, m_k\}$—allowing a single large commit to be decomposed into multiple modules Herzig et al. (2015)—via the following recursive logic:

1. **Initial Bound:** Compute the d_{LCA} of the leaf nodes from the commit C.
2. **Capacity Check:** Calculate the total file count (N_{files}) in the subtree rooted at d_{LCA}.
3. **Decision:**
   - **Accept:** If $N_{files} \leq N_{max}$, the directory d_{LCA} is a valid module.
   - **Split:** If $N_{files} > N_{max}$ *and* d_{LCA} contains subdirectories, we partition the files based on their immediate subdirectories and recursively apply Step 1 to each subset.
   - **Force Termination:** If $N_{files} > N_{max}$ but no further subdirectories exist, we accept d_{LCA} as a module, relying on the retrieval system to handle the excess context.

4.2 Topological Labeling

4.2.1 Semantic Consistency Filtering

Raw commit logs are inherently noisy, often characterized by generic updates (e.g., "fix bug", "update logic") that lack sufficient semantic signal to predict associated code changes Liu et al. (2018) inducing hallucination. To mitigate this, a **Semantic Consistency Filter** is employed to enforce alignment between the natural language intent and the structural output. Specifically, the concatenated context $x = [feature; module; file_paths]$ and the target task description y are embedded into a shared latent space via `all-MiniLM-L6-v2` Wang et al. (2020). The alignment is quantified as the cosine similarity score $S_{\text{sem}}(x, y)$:

$$S_{\text{sem}}(x, y) = \frac{E(x) \cdot E(y)}{\|E(x)\|\|E(y)\|} \tag{3}$$

Trajectories where $S_{\text{sem}} < \tau$ (set to 0.6 in experiments) are subsequently discarded. This thresholding eliminates $\approx 15\%$ of the raw corpus, effectively pruning "silent" commits or vague messages that lack predictive causality. For the detailed training prompt see Appendix C.

4.2.2 Concept-Balanced Temporal Splitting

Standard random splitting is unsuitable for software evolution because it violates temporal causality and introduces data leakage Allamanis (2018). Conversely, a strict chronological split often creates a distribution shift that undermines model performance. Although recent work emphasizes the importance of realistic labeling Jimenez et al. (2019), the distribution shift remains a critical challenge.

To resolve this dilemma, **Concept-Balanced Temporal Splitting** is implemented:

1. **Latent Clustering:** All feature requests are encoded into embeddings using `jina-embeddings-v2` Günther et al. (2023), followed by K-Means clustering ($K = 400$) to identify distinct functional concepts (e.g., "Authentication", "Data Pipeline", "UI Widget").
2. **Intra-Cluster Split:** Within each semantic cluster C_k, a strict temporal cutoff ($T_{\text{cut}} =$ Jan 1, 2025) is enforced:
   - Samples with $t < T_{\text{cut}}$ are assigned to **Train/Validation**.
   - Samples with $t \geq T_{\text{cut}}$ are assigned to **Evaluation**.

This protocol ensures that the model is evaluated on *future* data (preventing leakage) while guaranteeing that the evaluation set covers the same distribution of functional concepts as the training set.

4.3 Preference Pair Construction

To train the HDPO discriminator, we construct preference pairs (y_w, y_l) that force the model to distinguish between valid architectural reasoning and plausible but incorrect planning, following the standard preference optimization framework Rafailov et al. (2023b).

4.3.1 Synthesizing the Chosen Trajectory (y_w): TEACHER-GUIDED REASONING & CONTEXT SIMULATION

Since raw repository data provides ground-truth code (y_{gold}) but lacks the intermediate reasoning chain (z), we employ a **Teacher-Guided Synthesis** pipeline to reverse-engineer the architectural justification (z_{syn}) directly from the historical commit, anchoring the logic in the codebase topology Magister et al. (2023); Zelikman et al. (2022) (see Appendix D).

To prepare the model for inference-time retrieval noise, we employ an **Oracle-Augmented Context Simulation** strategy Zhang et al. (2024a); Izacard & Grave (2020). Although empirical analysis indicates that fewer than **3%** of features exceed standard context bounds, we explicitly enforce fixed windows by capping the Module Layer (L1) at $|T \cup D| = 50$ modules and the File Layer(L3) at $N = 60$ file paths.

Furthermore, while the Module Layer (L1) and File Layer (L3) layers utilize an enforced selection process Zhang et al. (2024a); Izacard & Grave (2020), the Task Layer (L2) remains a purely generative process. The detailed prompt templates are provided in Appendix **??**.

4.3.2 Synthesizing the Rejected Trajectory (y_l): HYBRID HARD-NEGATIVE SAMPLING

To rigorously penalize structural errors, the rejected trajectory y_l is sampled from a hybrid distribution designed to cover both "near-miss" errors and stochastic failures:

- **On-Policy Hard Negatives (60%):** The majority of rejected samples are generated by the base Step-SFT model itself ($\hat{y} \sim \pi_{\text{SFT}}(y|x)$). These represent "plausible hallucinations" Yuan et al. (2023)—errors that the model is statistically prone to making, such as selecting a deprecated module that semantically resembles the target.
- **Stochastic Shift Perturbations (40%):** The remaining portion consists of random noise injection where ground-truth targets are swapped with irrelevant candidates. This prevents the reward model from overfitting to the specific error distribution of the Step-SFT policy (reward hacking) and improves generalization to unseen inputs Touvron et al. (2023).

5 Experiment

5.1 Experimental Setup

We compare HDPO against five distinct paradigms to isolate the impact of hierarchical alignment:

- Step-SFT: Mistral-7B-Instruct-v0.2 Jiang et al. (2023) fine-tuned via cross-entropy loss, representing unweighted surface pattern matching.
- Step-Weighted-SFT (4:2:1): Fine-tuned with hierarchical cross-entropy weights (4:2:1), testing if supervised token re-weighting alone suffices without preference optimization.
- Step-DPO: Aligned with standard Step-DPO Lai et al. (2024) ($\alpha_t = 1$), testing whether flat preference optimization resolves hierarchical bottlenecks.
- RAG-based Selection: A retrieval baseline retrieving historical candidates for Modules (L1) and Files (L3), ranking Tasks (L2) via context Gao et al. (2023).
- Long-Context Frontier (Gemini-3.1-pro-preview): A zero-shot frontier baseline Comanici et al. (2025) testing whether intrinsic pre-training code knowledge can substitute for structural alignment.
- HDPO (Ours): Our framework enforcing gradient parity across architectural layers via structure-aware implicit rewards without inference overhead.

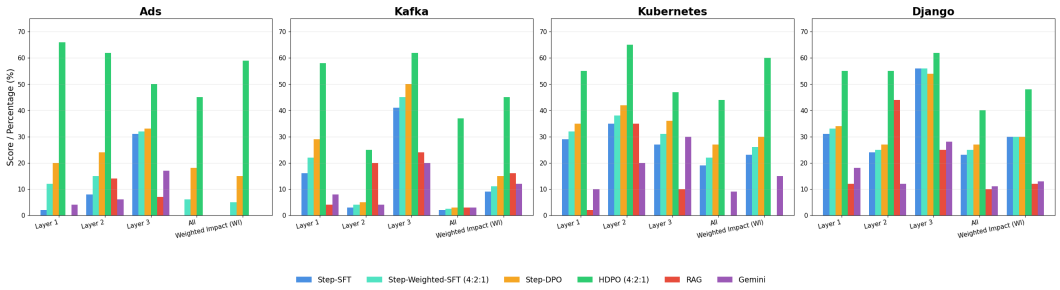


Figure 2: **Comparative Performance and Weighted Impact Analysis.** We evaluate success rates across Module (L1), Task (L2), and File (L3) layers alongside Weighted Impact (WI). **(1) Semantic Shortcut:** Baselines (Step-SFT, Step-Weighted-SFT, Step-DPO) display a visible disjoint in complex codebases; achieving partial Implementation (L3) success while failing Strategic (L1) planning. **(2) HDPO Dominance:** HDPO (4:2:1) (Green) resolves this structural collapse, achieving consistent End-to-End accuracy ($> 40\%$). **(3) Impact Gap:** Crucially, HDPO dominates the WI metric (far right bars), demonstrating that it solves complex, high-duration features where RAG (Red) and Gemini (Purple) fail due to reliance on retrieval shortcuts and parametric memory.

Implementation Details. All experiments utilize **Mistral-7B-Instruct-v0.2** Jiang et al. (2023) as backbone. Training was conducted on $8\times$ NVIDIA A100-80GB GPUs using DeepSpeed ZeRO-3 Rajbhandari et al. (2019) and LoRA Hu et al. (2021) ($r = 64$, $\alpha = 16$, $p = 0.05$) across linear layers. Models were trained for **1 epoch** with effective batch size 100 and a cosine schedule starting at 2e-5.

Test Benchmarks. We evaluate cross-domain robustness across four codebases spanning 2,500 feature requests:

- **Ads Domain (Proprietary)**: An industrial monorepo ($>$ 10M LOC); features average 5 modules and $>$ 3 tasks, peaking at 40 modules and 50 tasks.
- **Django (Python)**: A dynamic web framework with loosely coupled MVT patterns Holovaty & Kaplan-Moss (2007); 96% of requests reside in single modules.
- **Apache Kafka (Java)**: A distributed event log platform Kreps (2011) exhibiting moderate complexity (averaging 2 modules and 2+ tasks).
- **Kubernetes (Go)**: A container orchestration platform Burns et al. (2016) with deep package trees (averaging 2 modules and 1 task per feature).

Evaluation Metrics.

- Deterministic Matching (L1 & L3): Binary **Exact Match (EM)** against ground-truth module directories and file paths.
- Semantic Equivalence (L2): Evaluated using an **LLM-as-a-Judge** (`gemini-2.5-flash`) scoring plan semantics and topological order Zheng et al. (2023) (Appendix E).
- **End-to-End Success:** A request is resolved if and only if *all three layers* (L1, L2, L3) are simultaneously correct.

See Appendix F.1 for full per-domain tabular results.

5.2 Efficacy and Algorithmic Robustness

To quantify practical deployment impact, we evaluate using **Weighted Impact (WI)** Castillo et al. (2024)—an **industry-standard** metric adopted by EngProd teams that treats feature duration as a proxy for structural task complexity (Appendix B):

$$\text{WI} = \frac{\sum_{f \in S} \text{Duration}(f)}{\sum_{f \in \mathcal{D}} \text{Duration}(f)} \tag{4}$$

where S is the subset of resolved features and $\mathcal{D}$ is the complete evaluation set.

Visualizing the "Staircase" Failure Mode (Fig. 2). Across complex multi-module codebases (Ads and Kafka), STEP-SFT and STEP-DPO display an asymmetrical "ascending staircase" from L1 to L3: achieving moderate File accuracy (31.0% and 33.0% on Ads) while collapsing at Module routing (2.0% and 20.0%). Because implementation tokens vastly outnumber module identifiers ($|\mathcal{I}| \gg |\mathcal{S}|$), standard sequence objectives minimize token perplexity on low-level boilerplate while guessing module boundaries. This upstream failure cascades through the dependency DAG, reducing End-to-End accuracy (All) to 0.0% for Step-SFT and 18.0% for Step-DPO.

The Limits of Supervised Loss Re-Weighting. STEP-WEIGHTED-SFT (4:2:1) (Mint) tests whether manual token re-scaling in maximum-likelihood estimation (MLE) suffices. While it offers modest gains at Layer 1 on Ads (12.0% vs. 2.0%), performance stalls at Task planning (15.0%) and drops to 6.0% End-to-End (5.0% WI). Cross-entropy training merely boosts probability on positive examples without building an explicit contrastive margin against unviable paths, failing to suppress the sheer volume of implementation tokens.

Failure of Parametric Retrieval and Generalist Reasoning. External retrieval (RAG, Red) and generalist models (GEMINI, Purple) fail on enterprise routing. On Ads, RAG yields 0.0% on L1 and End-to-End accuracy, succeeding only on isolated file lookups (14.0% on L2, 7.0% on L3). Gemini similarly achieves only 4.0% on L1 and collapses to 0.0% End-to-End. While both show moderate performance on open-source Django due to pre-training exposure, their failure on Ads proves that parametric scale cannot substitute for structured architectural alignment.

HDPO Dominance and the WI $>$ Accuracy Inversion. HDPO (4:2:1) (Green) resolves this bottleneck, elevating Layer 1 routing to 66.0% on Ads ($+46.0\%$ over Step-DPO) and achieving 45.0% End-to-End accuracy. Crucially, as seen in the rightmost bars of Figure 2, **WI substantially exceeds End-to-End Accuracy for HDPO** (59.0% vs. 45.0% on Ads; 60.0% vs. 44.0% on Kubernetes). Baselines exhibit the reverse trend, resolving only trivial single-file fixes. This inversion demonstrates that HDPO successfully targets the high-effort, multi-module features that dominate developer time. By resolving early architectural branching points correctly, HDPO eliminates costly cross-module rework cycles that otherwise paralyze enterprise engineering workflows.

5.3 IMPACT OF HIERARCHICAL WEIGHT CONFIGURATIONS

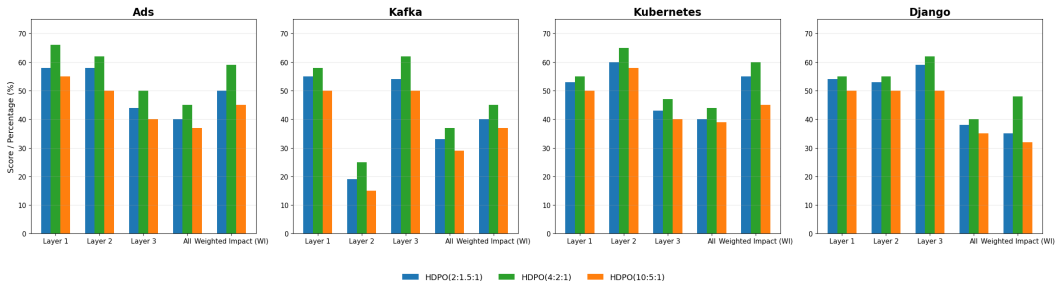
We analyze model sensitivity across three schedules: Conservative (λ = 2:1.5:1), Balanced (λ = 4:2:1), and Aggressive (λ = 10:5:1) (Figure 3a; protocol in Appendix A).

The Inverted-U Optimality Dynamics (Fig. 3a). Figure 3a confirms an "Inverted-U" performance profile across all benchmarks, with the **Balanced Configuration (Green,** 4:2:1**)** establishing the peak across all metrics:

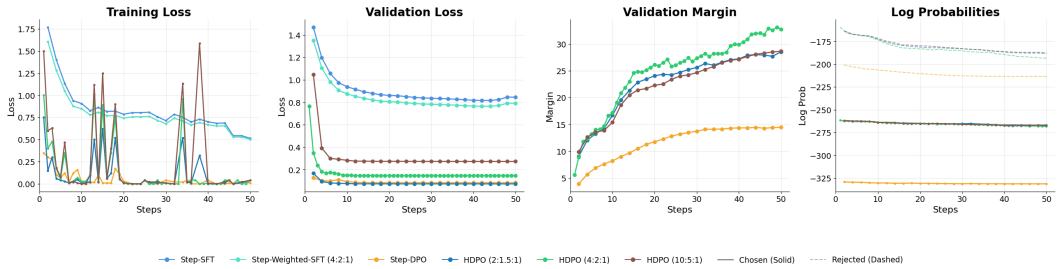
- **Under-Constrained Regime (Blue,** 2:1.5:1**):** Lighter penalties under-correct the Sparsity-Dilution pathology. File tokens dilute module signals, reducing L1 accuracy to 58.0% on Ads and limiting WI to 50.0% on Ads and 35.0% on Django.
- **Over-Constrained Regime (Orange,** 10:5:1**):** Excessive penalties ($\lambda = 10+$) degrade all benchmarks. On Ads, Layer 2 drops from 62.0% to 50.0%, and Layer 3 falls to 40.0%. This reflects *Gradient Stiffness*: excessive module weighting ($L_{\text{module}} \gg L_{\text{token}}$) impairs the model's plasticity for task planning and file localization.
- **Optimal Equilibrium (Green,** 4:2:1**):** Provides sufficient gradient amplification to counter the token length disparity without causing sigmoid saturation or gradient stiffness, maintaining high representational capacity across both coarse routing decisions and granular code synthesis.

5.4 TRAINING STABILITY AND OPTIMIZATION DYNAMICS

We examine 50-step optimization trajectories on Django (Figure 3b).



(a) **Ablation Study on Hierarchical Weights.** We compare Conservative (Blue, 2:1.5:1), Balanced (Green, 4:2:1), and Aggressive (Orange, 10:5:1) weight schedules across all four benchmarks. The Balanced configuration ($\lambda = 4{:}2{:}1$) consistently yields the highest Weighted Impact (WI) and End-to-End accuracy, confirming that while structural gradient enforcement is essential, excessive scaling ($\lambda = 10+$) induces gradient stiffness and over-constrains policy optimization.



(b) **Django Training Dynamics and Margin Analysis.** HDPO (4:2:1, Green) stabilizes rapidly relative to cross-entropy baselines (Step-SFT and Step-Weighted-SFT) and maintains smooth validation loss convergence alongside adjacent variants (2:1.5:1, 10:5:1). Crucially, HDPO (4:2:1) more than doubles the **Validation Margin** compared to Step-DPO (Orange, $\approx 14.5 \rightarrow 32.8$), demonstrating that hierarchical gradient alignment enforces substantial likelihood separation between ground-truth sequences and structurally invalid candidates.

Figure 3: **Parameter Sensitivity and Optimization Analysis.**

Convergence Velocity and Loss Disconnect (Panels 1 & 2). STEP-SFT (Blue) and STEP-WEIGHTED-SFT (Mint) plateau at high validation loss ($L \approx 0.82$–0.85). While STEP-DPO (Orange) reaches $L \approx 0.08$, it does so by fitting low-entropy syntax in Layer 3 rather than architectural routing. HDPO variants converge smoothly within 10 steps to well-calibrated loss plateaus ($L \approx 0.15$ for 4:2:1; $L \approx 0.28$ for 10:5:1), maintaining active gradients across all planning layers.

Discriminative Margin Maximization (Panel 3). Defining reward margin as $\mathcal{M} = \mathbb{E}[\log \pi_\theta(y_w) - \log \pi_\theta(y_l)]$, Step-DPO plateaus early at $\mathcal{M} \approx 14.5$. Conversely, HDPO variants expand steadily, with Balanced HDPO (4:2:1, Green) reaching $\mathbf{32.8}$ by step 50—more than doubling Step-DPO. This likelihood separation penalizes incorrect module trajectories and suppresses hallucinated plans.

Log-Probability Separation and Stability Cliffs (Panels 1 & 4). The policy maintains stable log-likelihood on chosen plans ($\log \pi_\theta(y_w) \approx -268$, solid lines) while depressing structurally invalid candidates ($\log \pi_\theta(y_l)$ drops from -160 to -193, dashed lines). Furthermore, Panel 1 reveals training loss spikes past 1.2–1.6 under the Aggressive schedule (10:5:1, Brown) due to gradient shocks at module mispredictions. Balanced HDPO (4:2:1, Green) eliminates these spikes, achieving stable convergence and optimal discriminative separation.

AI USE STATEMENT

In this work, we used generative AI tools (`gemini-2.5-flash`) for dataset synthesis (specifically reverse-engineering architectural rationale traces for the chosen trajectories, as described in Section 4.3.1 and Appendix D) and for automated execution-plan evaluation under an LLM-as-a-Judge protocol (Appendix E). We have not used generative AI tools for core algorithm conception, mathematical

derivations, or initial experimental code development. Additionally, we used generative AI tools for writing assistance, limited to proofreading and grammatical polishing. We have thoroughly reviewed all AI-assisted work: the synthetic reasoning traces were audited via manual human review across a 10% stratified sample (yielding a 99.0% agreement rate), and the LLM judge decisions were independently validated against human expert consensus (achieving 96.0% decision alignment). We take full responsibility for the final content of this work, including all empirical claims, mathematical formulations, and textual artifacts produced with the aid of generative AI.

Ethics statement

Our research strictly adheres to the ICLR Code of Ethics. The empirical evaluation utilizes publicly available open-source repositories (Django, Apache Kafka, and Kubernetes), which are licensed under permissive open-source licenses (BSD 3-Clause and Apache 2.0). All mined commits, diffs, and pull requests adhere to the original licensing terms and usage policies. For the industrial proprietary benchmark (Ads Domain), all telemetry, commit logs, author identities, and repository paths were fully anonymized and sanitized prior to experimentation to protect organizational privacy and intellectual property; no personally identifiable information (PII) or confidential client data was accessed or retained. Human validation studies involved two consenting professional software engineers performing expert code audits; no external or vulnerable human subjects were involved, and no compensation-related conflicts exist. While improving automated software planning may influence engineering workflows, our methodology is designed to augment developers by resolving macro-architectural bottlenecks rather than displacing human oversight in security-critical codebases.

Reproducibility statement

To ensure the reproducibility of our methodology and empirical results, we have documented the theoretical formulation, algorithmic implementation, and dataset curation processes throughout the manuscript and its appendices. Specifically, the Hierarchical Direct Preference Optimization (HDPO) algorithm is formalized in Section 3.1 and Algorithm 1. Complete training hyperparameters, infrastructure details, and baseline configurations are provided in Section 5.1. To facilitate the reproduction of our Hierarchical Evolution Dataset (HED), Section 4 details the dynamic complexity-adaptive decomposition and splitting protocol, while Appendix C and Appendix D provide the exact prompt templates and the Teacher-Guided Synthesis methodology used for reasoning generation. The comprehensive LLM-as-a-Judge evaluation rubric is outlined in Appendix E, and our rigorous statistical reporting protocol—including the computation of 95% confidence intervals across independent runs—is detailed in Appendix F.2. Finally, to support independent verification, the declared gain is based on the open-source projects (Django, Kafka, Kubernetes).

References

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 143–153. ACM, 2018. doi: 10.1145/3359591.3359735.

Saleema Amershi, Andrew Begel, Christian Bird, R. DeLine, H. Gall, Ece Kamar, N. Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 291–300. IEEE, IEEE, 2019. doi: 10.1109/ICSE-SEIP.2019.00042.

Carlos Arguelles. How we use hermetic, ephemeral test environments at google to reduce test flakiness. In *International Conference on Software Testing, Verification and Validation Workshops*, pp. 99–99. IEEE, IEEE, 2023. doi: 10.1109/icstw58534.2023.00029.

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*, pp. 1123–1132. IEEE, 2017. doi: 10.1109/BigData.2017.8258038.

Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016. doi: 10.1145/2890784.

Yadira Jazmín Pérez Castillo, S. D. O. Jiménez, and P. Torres. Sprint management in agile approach: Progress and velocity evaluation applying machine learning. *Inf.*, 15:726, 2024. doi: 10.3390/info15110726.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv.org*, 2021.

P. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, S. Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Neural Information Processing Systems*, 2017.

Gheorghe Comanici, E. Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit S. Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv.org*, 2025.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Hua-Zuo Gao, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv.org*, 2024. doi: 10.48550/arXiv.2406.11931.

Premkumar T. Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, pp. 837–847, 2012. doi: 10.1109/ICSE.2012.6227135.

N. Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, M. Mazzara, F. Montesi, Ruslan Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pp. 195–216, 2016. doi: 10.1007/978-3-319-67425-4_12.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Jianwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *International Conference on Software Engineering*, pp. 1–13. ACM, 2024. doi: 10.1145/3597503.3639219.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, A. Chao, Apurva N. Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv.org*, 2024.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kang-Xiang Jia, Jin Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qian-Yu Guo, Meng Wang, et al. Retrieval-augmented generation for large language models: A survey. *arXiv.org*, 2023.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, et al. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *arXiv.org*, 2023.

Kim Herzig, Sascha Just, and A. Zeller. The impact of tangled code changes on defect prediction. In *Empirical Software Engineering*, volume 21, pp. 303–336. Springer Science and Business Media LLC, 2015. doi: 10.1007/s10664-015-9376-6.

Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right*. Apress, 2007.

Jiwoo Hong, Noah Lee, and James Thorne. Orpo: Monolithic preference optimization without reference model. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 11170–11189. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.626.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Z. Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. *International Conference on Learning Representations*, 2023.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

Gautier Izacard and Edouard Grave. Distilling knowledge from reader to retriever for question answering. In *International Conference on Learning Representations*, 2020.

Ciera Jaspan and Collin Green. Measuring productivity: All models are wrong but some are useful. *IEEE Software*, 42(2):13–17, 2025. doi: 10.1109/ms.2024.3511735.

Albert Qiaochu Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv.org*, 2023. doi: 10.48550/arXiv.2310.06825.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shun-Yu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *International Conference on Learning Representations*, 2023. doi: 10.48550/arXiv.2310.06770.

Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Y. Traon, and M. Harman. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *ESEC/SIGSOFT FSE*, pp. 695–705. ACM, ACM, 2019. doi: 10.1145/3338906.3338941.

Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Knowledge Discovery and Data Mining*, pp. 1168–1176. ACM, ACM, 2013. doi: 10.1145/2487575.2488217.

J. Kreps. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pp. 1–7, 2011.

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xi Peng, and Jiaya Jia. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *arXiv.org*, 2024. doi: 10.48550/arXiv.2406.18629.

Jia Li, Xuyuan Guo, Lei Li, Kechi Zhang, Ge Li, Zhengwei Tao, Fang Liu, Chongyang Tao, Yuqi Zhu, and Zhi Jin. Longcodeu: Benchmarking long-context language models on long code understanding. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27309–27327. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.1324.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2023a. doi: 10.1162/tacl_a_00638.

Tianyang Liu, Can Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *International Conference on Learning Representations*, 2023b. doi: 10.48550/arXiv.2306.03091.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. Neural-machine-translation-based commit message generation: how far are we? In *International Conference on Automated Software Engineering*, pp. 373–384. ACM, 2018. doi: 10.1145/3238147.3238190.

L. Magister, Jonathan Mallinson, Jakub Adamek, Eric Malmi, and A. Severyn. Teaching small language models to reason. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 1773–1781. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.acl-short.151.

Yu Meng, Mengzhou Xia, and Danqi Chen. Simpo: Simple preference optimization with a reference-free reward. In *Advances in Neural Information Processing Systems 37*, pp. 124198–124235. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024. doi: 10.52202/079017-3946.

Shaun Ossei-Owusu. Code red. *American Journal of Law Medicine*, 43(4):344–387, 2017. doi: 10.1177/0098858817753404.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, S. Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Neural Information Processing Systems*, volume 35, pp. 27730–27744. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022. doi: 10.52202/068431-2011.

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 *Neural Information Processing Systems*, volume 36, pp. 53728–53741. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023a. doi: 10.52202/075280-2338.

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 *Neural Information Processing Systems*, volume 36, pp. 53728–53741. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023b. doi: 10.52202/075280-2338.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, IEEE, 2019. doi: 10.1109/SC41405.2020.00024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv.org*, 2017.

David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Neural Information Processing Systems*, pp. 2503–2511, 2015.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Jun-Mei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv.org*, 2024. doi: 10.48550/arXiv.2402.03300.

Noah Shinn, Federico Cassano, Beck Labash, A. Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Neural Information Processing Systems*, pp. 8634–8652. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-0377.

Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Aditya Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Symposium on Operating Systems Principles*, pp. 328–343. ACM, ACM, 2015. doi: 10.1145/2815400.2815401.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, X. Martinet, M. Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv.org*, 2023.

Miles Turpin, Julian Michael, Ethan Perez, and Sam Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems 36*, pp. 74952–74965, 2023. doi: 10.52202/075280-3275.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Neural Information Processing Systems*, 2020.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *International Conference on Learning Representations*, 2024.

Muhammad Waseem, Peng Liang, Aakash Ahmad, A. Khan, Mojtaba Shahin, A. R. Nasab, T. Mikkonen, and P. Abrahamsson. Understanding the issues, their causes and solutions in microservices systems: An empirical study. *Journal of Systems and Software*, 237:112828, 2026. doi: 10.48550/arXiv.2302.01894.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, F. Xia, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Neural Information Processing Systems*, pp. 24824–24837. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022. doi: 10.52202/068431-1800.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems 38*, pp. 87218–87243, 2025. doi: 10.52202/085713-2629. URL https://arxiv.org/abs/2502.18449.

Huizhi Xie and Juliette Aurisset. Improving the sensitivity of online controlled experiments: Case studies at netflix and microsoft. In *Knowledge Discovery and Data Mining*, pp. 50–58. ACM, 2016. doi: 10.1145/2939672.2939733.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Adriano Lieret, Shun-Yu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems 37*, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, T. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems 36*, pp. 11809–11822. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-0517.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. Rrhf: Rank responses to align language models with human feedback without tears. In *Advances in Neural Information Processing Systems 36*, volume 36, pp. 10935–10950. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-0482.

E. Zelikman, Yuhuai Wu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems 35*, volume 35, pp. 15476–15488. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022. doi: 10.52202/068431-1126.

Yong-Cheng Zeng, Guo-Qing Liu, Weiyu Ma, Ning Yang, Haifeng Zhang, and Jun Wang. Token-level direct preference optimization. In *International Conference on Machine Learning*, 2024. doi: 10.48550/arXiv.2404.11999. URL https://arxiv.org/abs/2404.11999.

Tianjun Zhang, Shishir G. Patil, Naman Jain, Sheng Shen, M. Zaharia, Ion Stoica, and Joseph E. Gonzalez. Raft: Adapting language model to domain specific rag. *arXiv.org*, 2024a. doi: 10.48550/arXiv.2403.10131.

Yun-Tong Zhang, Haifeng Ruan, Zhi-Yu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *International Symposium on Software Testing and Analysis*, pp. 1592–1604. ACM, 2024b. doi: 10.1145/3650212.3680384.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Da-Cheng Li, Eric P. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Neural Information Processing Systems*, volume 36, pp. 46595–46623. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-2020.

A Design, Validation, and Robustness of the 4:2:1 Weight Schedule

A.0.1 Theoretical Result: The Sparsity-Dilution Pathology

Consider the standard DPO gradient. The total parameter update is the summation of contributions from the Module layer ($\mathcal{S}$) and the Filepath layer ($\mathcal{I}$):

$$\nabla_\theta \mathcal{L}_{\text{DPO}} = \sum_{t \in \mathcal{S}} \nabla_\theta \ell_t + \sum_{t \in \mathcal{I}} \nabla_\theta \ell_t \tag{5}$$

In software engineering tasks, the sequence distribution is heavily skewed. The Module layer constitutes $< 1\%$ of the sequence length ($|\mathcal{S}| \ll |\mathcal{I}|$), yet contains 100% of the structural failure modes. Consequently, the magnitude of the Filepath layer gradients completely dominates the Step-DPO update vector:

$$\left\| \sum_{t \in \mathcal{I}} \nabla_\theta \ell_t \right\| \gg \left\| \sum_{t \in \mathcal{S}} \nabla_\theta \ell_t \right\| \tag{6}$$

Our theoretical finding: This imbalance induces a vanishing reasoning signal. The optimizer minimizes the loss by refining thousands of narrative tokens while leaving fatal structural errors uncorrected, effectively training the model to be "fluent but wrong."

A.0.2 New Loss Formulation: Enforcing Gradient Parity

To resolve this pathology, HDPO reformulates the objective by introducing an importance scalar λ. Rather than serving as an arbitrary scaling factor, λ is mathematically derived to enforce approximate parity between the gradient magnitude of the Module layer and that of the Filepath layer:

$$\mathbb{E}\left[\left\| \sum_{t \in \mathcal{S}} \lambda_{\text{module}} \nabla_\theta \ell_t \right\| \right] \approx \mathbb{E}\left[\left\| \sum_{t \in \mathcal{I}} \nabla_\theta \ell_t \right\| \right] \tag{7}$$

Empirical measurements of Filepath layer gradient norms during early standard DPO epochs demonstrate that unweighted Filepath tokens dominate Module tokens by a factor of $15\times$ to $20\times$. While our loss formulation is specifically designed to constrain this gradient imbalance, setting $\lambda = 20$ directly is prohibited by two algorithmic constraints:

- **Magnitude Ceiling (Absolute Scale):** Excessively high absolute weights (≥ 20) cause accumulated rewards to saturate the DPO sigmoid loss, resulting in vanishing gradients and optimization failure.
- **Ratio Balance (Relative Scale):** The optimization schedule must identify the steepest inter-layer ratio that operates strictly beneath this saturation ceiling to reliably restore gradient parity.

A.0.3 Balanced Configuration Search and Robustness

We conducted a two-phase orthogonal search to establish the empirical equilibrium on a 20% held-out validation set.

- **Phase 1: Binary Search for the Magnitude Ceiling (Fixed Ratio):** Fixing the inter-layer ratio at $\sim 2\times$, we performed an open-ended magnitude sweep. An initial theoretical probe at $100 : 50 : 25$ failed immediately due to sigmoid saturation, establishing an empirical upper bound. Conversely, a conservative lower-bound probe at $4 : 2 : 1$ converged stably. Halving the search interval ($100 \to 50 \to 20 \to 10$) revealed that $50 : 25 : 12$ and $20 : 10 : 5$ also failed to converge, locating the operational ceiling at $10 : 5 : 1$. Evaluating the viable candidates ($10 : 5 : 1$, $4 : 2 : 1$, $2 : 1 : 0.5$) indicated that a base magnitude of 4 provided the optimal trade-off between gradient flow and structural enforcement.
- **Phase 2: Verifying Ratio Stability (Fixed Magnitude):** To verify that convergence is governed by absolute scale rather than an isolated ratio artifact, we fixed the base magnitude

at 4 and evaluated adjacent spreads ($4:3:1$, $4:2:1$, and $4:1.5:1$). Consistent with theoretical expectations, all three configurations converged smoothly, with $4:2:1$ achieving the highest downstream Accuracy and Weighted Impact (WPI) on the held-out split.

Consequently, our experimental evaluation presents $10:5:1$, $2:1:0.5$, and $4:2:1$ to systematically delineate the mathematically viable search landscape:

- **10:5:1** marks the upper convergence ceiling, inducing *Gradient Stiffness*.
- **2:1:0.5** serves as an under-constrained lower bound, permitting *Semantic Shortcuts*.
- **4:2:1** establishes the optimal empirical equilibrium.

B Industry Background and Evaluation Disconnect

B.1 Code Quality (pass@k) Is Not the Bottleneck in Complex Feature Planning

In the software development lifecycle, a fundamental dichotomy exists between coding quality and coding efficiency:

- **Code Quality:** Metrics-driven and evaluated via isolated unit and integration test pass rates (typically parameterized as pass@k). This localized functional correctness forms the primary focus of prevailing LLM code generation research.
- **Coding Efficiency (Feature Velocity):** The engineering time and effort expended *after* isolated code passes standard unit and integration tests. In enterprise feature development, engineers routinely implement isolated components with high speed once provided an explicit task specification. The primary bottleneck lies instead in discovering *what* must be implemented across interdependent modules when individual teams only comprehend their local subsystems and comprehensive multi-module test suites are absent.

The Debug-on-Fail Loop As illustrated in the feature development lifecycle (Figure 1), the most resource-intensive "debug-on-fail" loops emerge during concurrent cross-module integration. Crucially, these failures surface *after* the local process of clearing unit tests, submitting patches, and passing initial continuous integration (CI/CD) checks. Because standard pipelines validate components in isolation, authoring code within a team's own boundary is rarely the gating constraint, supported by direct test feedback. The breakdown occurs because reasoning over concurrent cross-module dependencies exceeds a single team's cognitive bandwidth, leaving developers blind to upstream and downstream structural obligations.

The Industry Status Quo This dynamic represents an entrenched structural challenge. Software organizations logically partition engineering organizations into domain-specific silos (e.g., Client, Data, Infrastructure) to constrain localized domain complexity. However, this organizational boundary creates substantial friction for cross-cutting feature delivery. To mitigate this, enterprises deploy dedicated Engineering Productivity (EngProd) organizations to construct localized staging and integration suites. In practice, these testing harnesses require quarters of engineering investment to maintain, cover only a small subset of interdependent components due to data and infrastructure complexity, and introduce heavyweight configurations and slow runtimes merely to determine cross-module task scope.

The HDPO Formulation: Frontloading Task Scoping HDPO operates on the premise that while component-level synthesis can be achieved with high functional fidelity, feature delivery stalls due to an absence of global architectural scoping. Regardless of intervening development iterations, the final merged "chain-of-patches" represents the complete, ground-truth dependency graph: an explicit DAG of required changes. HDPO leverages this historical trajectory to generate the complete cross-module scope upfront. By eliminating late-stage debug-on-fail cycles, HDPO supplies engineers with an end-to-end task blueprint from initialization, preserving established, high-throughput localized implementation practices.

B.2 METRIC CLARIFICATION: WEIGHTED IMPACT (WI)

Evaluating full-scope delivery requires an efficiency metric aligned with real-world developer economics. Standard EngProd evaluations measuring tooling that automates cross-module discovery rely on *Weighted Impact* (WI):

$$\text{WI} = \frac{\sum_{f \in S} \text{Duration}(f)}{\sum_{f \in \mathcal{D}} \text{Duration}(f)} \tag{8}$$

where S denotes the subset of successfully resolved features, $\mathcal{D}$ denotes the full evaluation suite, and $\text{Duration}(f)$ represents the historical resolution duration, serving as an empirical weight for the structural complexity of feature f. (Note: We clarify that the calculated values remain identical to the metric termed "development efficiency" in the original submission; this camera-ready revision formalizes the notation and terminology.)

Justifying the Theoretical Upper Bound While WI assumes an upper bound on potential duration savings (as not all human activity is optimizable), metrics-driven organizations (e.g., large-scale ad-tech and search platforms) rely on this formulation because perfectly granular developer time-tracking is intractable. Empirical studies demonstrate that software development involves frequent task switching and discontinuous workflows. Raw timestamp deltas across commits, tickets, and reviews capture asynchronous context switches, meeting overhead, and operational pauses rather than continuous implementation time. Consequently, end-to-end resolution duration serves as the standard, reproducible proxy for systemic task complexity.

C TRAINING PROMPT CONSTRUCTION

To enforce structured, deterministic, and parseable generation across all three hierarchical layers of the HDPO framework, we enforce explicit persona-driven instructions alongside strict JSON schema constraints. The model is trained and evaluated using the following layer-specific templates:

- **Layer 1 (Module Routing):**

  "*You are a Senior Software Architect. Route the Feature Request to the correct Codebase Modules. Choose ONLY from the 'Available Modules' list. Output valid JSON with keys:* `reasoning` *and* `modules`."

- **Layer 2 (Task Decomposition):**

  "*You are a Senior Software Architect. Create a detailed, ordered execution plan. Explain your reasoning for the plan's content and order. Output strictly valid JSON with keys:* `reasoning_about_plan_content`, `reasoning_about_plan_order`, *and* `plan_steps` *(a list of strings representing the actionable steps).*"

- **Layer 3 (File Path Resolution):**

  "*You are a Senior Software Developer. Select the required* `file_paths` *from the module files listed above. Output valid JSON with keys:* `reasoning` *and* `file_paths`."

D REASONING DATA SYNTHESIS

D.1 TEACHER-DRIVEN CHAIN-OF-THOUGHT GENERATION (`GEMINI-2.5-FLASH`)

While historical software repositories (e.g., merged pull requests, commit trees, and issue resolutions) provide exact ground-truth targets for modified modules, sequential tasks, and touched file paths, they do not inherently record the intermediate architectural rationale required for supervised fine-tuning and Chain-of-Thought (CoT) alignment.

To bridge this supervision gap, we deploy a teacher model, `gemini-2.5-flash`, to back-translate merged historical changes into step-by-step reasoning traces conditioned directly on repository artifacts:

- **Layer 1 (Module Reasoning):** Conditioned on the feature description and codebase architecture, the model synthesizes the dependency reasoning explaining why the historically modified modules were selected over candidate alternatives.
- **Layer 2 (Task Reasoning):** Conditioned on the historical sequence of patches, the model generates dual-stage rationales establishing both the justification for task content and the causal execution dependencies governing task order.
- **Layer 3 (File Path Reasoning):** Conditioned on repository git diffs, the model explains why specific file paths within the selected module boundaries require modification to satisfy the upstream tasks.

D.2 Human Validation and Quality Assurance

Because the synthesized rationales are conditioned on factual historical commits, we verified the factual alignment of the generated training and evaluation corpora via a manual human audit:

- **Sampling Protocol:** Two senior software engineers independently reviewed a randomly sampled 10% subset of synthesized reasoning traces across the training, validation, and evaluation splits.
- **Evaluation Criterion (Reasoning Faithfulness):** Evaluators assessed whether the generated rationale remained strictly grounded in the historical code changes and repository DAG, penalizing hallucinations or ungrounded technical justifications.
- **Audit Findings:** The evaluation demonstrated a False Positive (FP) rate of 2.0% and an overall Human–LLM agreement rate of 99.0%, verifying that the generated rationales faithfully explain the historical ground truth.

E LLM-as-a-Judge Evaluation

E.1 Evaluation Protocol (`gemini-2.5-flash`)

Standard string-overlap metrics (e.g., BLEU, ROUGE) are inadequate for assessing Layer 2 execution plans, as valid plans frequently use semantic synonyms or alternate syntactic structures. We implement an automated LLM-as-a-Judge protocol using `gemini-2.5-flash` under a "Senior QA Engineer" persona. The judge compares candidate predictions against historical human ground truth across two orthogonal dimensions:

1. **Content Match (Permutation-Invariant):**
   - **FULL:** Every canonical step in the reference plan is accounted for in the prediction (semantic equivalents accepted).
   - **PARTIAL:** A subset of required steps is present, but critical implementation steps are missing or incorrect.
   - **NONE:** No semantic overlap with the reference specification.
2. **Order Match (Evaluated Over Matched Steps):**
   - **YES:** Matched steps adhere to the valid topological execution order defined by the ground truth.
   - **NO:** The relative sequence contains inverted prerequisites or illogical transitions.
   - **N/A:** Graded as non-applicable when Content Match is **NONE**.

E.2 Human Validation of Judge Verdicts

To ensure the automated judge produced reliable, reproducible grades, two senior software engineers conducted a stratified audit on a 10% random sample of the evaluation verdicts:

- **Reasoning Faithfulness (+1):** Evaluated whether the critique produced by the judge accurately characterized the differences between the prediction and reference without hallucinating missing requirements.

- **Decision Alignment (+1):** Evaluated whether the final categorical label (FULL, PARTIAL, YES, etc.) followed directly and logically from the critique.
- **Audit Findings:** The judge demonstrated an FP rate of 1.5%, an FN rate of 2.3%, and an aggregate Human–LLM decision agreement rate of 96.0%, confirming that automated judgments accurately reflect human expert scoring.

E.3 Mitigation of Same-Model Bias

Employing `gemini-2.5-flash` for both rationale generation (Appendix D) and judge evaluation introduces potential circularity or same-model stylistic preference. We mitigate this through two design constraints:

1. **Grounding to Human Ground Truth:** The judge evaluates predicted outputs strictly against the *historical human commit sequence*, rather than model-generated rationales or synthetic reference plans.
2. **Discrete Categorical Bottleneck:** Rather than using open-ended Likert scales or numerical scoring (which are vulnerable to stylistic and verbosity biases), the evaluation enforces a discrete rubric (FULL / PARTIAL / NONE and YES / NO) bounded by binary operational checks.

F Experimental Results: Robustness, Confidence Intervals, and Reproducibility

F.1 Benchmark Evaluation and Quantitative Comparison

Table 1 presents the quantitative evaluation across four representative software engineering domains: an enterprise proprietary codebase (**Ads**) and three open-source distributed frameworks (**Kafka**, **Kubernetes**, and **Django**). We benchmark HDPO variants against supervised fine-tuning baselines (Step-SFT, Step-Weighted-SFT), retrieval-augmented generation (RAG), standard sequence-level preference optimization (Step-DPO), and frontier instruction models (`gemini-2.5-pro` and `gemini-3.1-pro-preview`).

F.2 Confidence Intervals and Reproducibility Protocol

To ensure statistical rigor and evaluate run-to-run variance, every reported metric in Table 1 represents the mean across $n = 3$ independent runs:

- **Local Models:** All supervised fine-tuning variants (Step-SFT, Step-Weighted-SFT), canonical sequence-level DPO (Step-DPO), and hierarchical variants (HDPO) were initialized, trained, and evaluated over 3 independent trials with randomized seeds.
- **External and API Baselines:** Retrieval-augmented pipelines (RAG), `gemini-2.5-pro`, and `gemini-3.1-pro-preview` were queried across 3 discrete evaluation passes to capture decoding stochasticity and sampling drift.
- **Variance Quantification:** We report the 95% Confidence Interval (95% CI) calculated as:

$$\text{CI}_{95} = \bar{x} \pm 1.96 \cdot \frac{s}{\sqrt{n}} \tag{9}$$

where $\bar{x}$ denotes the empirical sample mean, s is the unbiased sample standard deviation, and $n = 3$. Deterministic retrieval baselines with zero run-to-run sampling variance report ± 0.0.

Table 1: Cross-domain evaluation across hierarchical layers, End-to-End accuracy, and Weighted Impact (WI). Results are reported as mean $\pm$ 95% CI over 3 independent runs. Best results per domain are highlighted in **bold**.

Domain	Model	Layer 1 Acc. (%)	Layer 2 Acc. (%)	Layer 3 Acc. (%)	End-to-End Acc. (%)	Weighted Impact (WI)
	Step-SFT	2.0 ± 0.5	8.0 ± 1.2	31.0 ± 2.5	0.0 ± 0.0	0.0 ± 0.0
	Step-Weighted-SFT	12.0 ± 1.5	15.0 ± 1.8	32.0 ± 2.1	6.0 ± 1.0	5.0 ± 0.8
	RAG	0.0 ± 0.0	14.0 ± 0.0	7.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	Gemini-2.5-Pro	4.0 ± 0.4	6.0 ± 0.5	17.0 ± 0.8	0.0 ± 0.0	0.0 ± 0.0
Ads	gemini-3.1-pro-preview	5.5 ± 0.5	7.5 ± 0.6	19.5 ± 1.0	1.0 ± 0.2	0.5 ± 0.1
	Step-DPO	20.0 ± 2.1	24.0 ± 2.4	33.0 ± 2.6	18.0 ± 1.9	15.0 ± 1.7
	HDPO (10:5:1)	55.0 ± 2.3	50.0 ± 2.5	40.0 ± 2.1	37.0 ± 2.0	45.0 ± 2.4
	HDPO (2:1.5:1)	58.0 ± 2.1	58.0 ± 2.2	44.0 ± 2.4	40.0 ± 1.8	50.0 ± 2.1
	HDPO (4:2:1)	$\mathbf{66.0 \pm 1.8}$	$\mathbf{62.0 \pm 1.9}$	$\mathbf{50.0 \pm 1.7}$	$\mathbf{45.0 \pm 1.5}$	$\mathbf{59.0 \pm 1.6}$
	Step-SFT	16.0 ± 1.8	3.0 ± 0.6	41.0 ± 2.7	2.0 ± 0.5	9.0 ± 1.1
	Step-Weighted-SFT	22.0 ± 2.0	4.0 ± 0.8	45.0 ± 2.5	2.5 ± 0.6	11.0 ± 1.3
	RAG	4.0 ± 0.0	20.0 ± 0.0	24.0 ± 0.0	3.0 ± 0.0	16.0 ± 0.0
	Gemini-2.5-Pro	8.0 ± 0.6	4.0 ± 0.4	20.0 ± 0.9	3.0 ± 0.4	12.0 ± 0.7
Kafka	gemini-3.1-pro-preview	11.0 ± 0.8	6.5 ± 0.5	23.5 ± 1.2	4.5 ± 0.5	13.5 ± 0.9
	Step-DPO	29.0 ± 2.5	5.0 ± 0.9	50.0 ± 2.8	3.0 ± 0.7	15.0 ± 1.6
	HDPO (10:5:1)	50.0 ± 2.6	15.0 ± 1.8	50.0 ± 2.5	29.0 ± 2.2	37.0 ± 2.3
	HDPO (2:1.5:1)	55.0 ± 2.3	19.0 ± 2.0	54.0 ± 2.3	33.0 ± 2.1	40.0 ± 2.1
	HDPO (4:2:1)	$\mathbf{58.0 \pm 1.9}$	$\mathbf{25.0 \pm 2.1}$	$\mathbf{62.0 \pm 1.8}$	$\mathbf{37.0 \pm 1.9}$	$\mathbf{45.0 \pm 1.7}$
	Step-SFT	29.0 ± 2.4	35.0 ± 2.6	27.0 ± 2.2	19.0 ± 1.9	23.0 ± 2.1
	Step-Weighted-SFT	32.0 ± 2.5	38.0 ± 2.7	31.0 ± 2.4	22.0 ± 2.1	26.0 ± 2.2
	RAG	2.0 ± 0.0	35.0 ± 0.0	10.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
	Gemini-2.5-Pro	10.0 ± 0.7	20.0 ± 1.0	30.0 ± 1.1	9.0 ± 0.6	15.0 ± 0.8
Kubernetes	gemini-3.1-pro-preview	13.5 ± 0.9	24.0 ± 1.2	33.5 ± 1.4	11.5 ± 0.8	17.5 ± 1.0
	Step-DPO	35.0 ± 2.7	42.0 ± 2.8	36.0 ± 2.6	27.0 ± 2.4	30.0 ± 2.5
	HDPO (10:5:1)	50.0 ± 2.5	58.0 ± 2.7	40.0 ± 2.4	39.0 ± 2.3	45.0 ± 2.4
	HDPO (2:1.5:1)	53.0 ± 2.3	60.0 ± 2.5	43.0 ± 2.3	40.0 ± 2.1	55.0 ± 2.2
	HDPO (4:2:1)	$\mathbf{55.0 \pm 1.8}$	$\mathbf{65.0 \pm 1.7}$	$\mathbf{47.0 \pm 1.9}$	$\mathbf{44.0 \pm 1.8}$	$\mathbf{60.0 \pm 1.5}$
	Step-SFT	31.0 ± 2.5	24.0 ± 2.1	56.0 ± 2.7	23.0 ± 2.0	30.0 ± 2.4
	Step-Weighted-SFT	33.0 ± 2.6	25.0 ± 2.2	56.0 ± 2.6	25.0 ± 2.2	30.0 ± 2.3
	RAG	12.0 ± 0.0	44.0 ± 0.0	25.0 ± 0.0	10.0 ± 0.0	12.0 ± 0.0
	Gemini-2.5-Pro	18.0 ± 0.9	12.0 ± 0.7	28.0 ± 1.1	11.0 ± 0.6	13.0 ± 0.8
Django	gemini-3.1-pro-preview	21.5 ± 1.1	15.5 ± 0.9	31.5 ± 1.3	13.5 ± 0.7	15.5 ± 0.9
	Step-DPO	34.0 ± 2.6	27.0 ± 2.3	54.0 ± 2.5	27.0 ± 2.3	30.0 ± 2.4
	HDPO (10:5:1)	50.0 ± 2.8	50.0 ± 2.8	50.0 ± 2.7	35.0 ± 2.5	32.0 ± 2.4
	HDPO (2:1.5:1)	54.0 ± 2.4	53.0 ± 2.5	59.0 ± 2.4	38.0 ± 2.2	35.0 ± 2.3
	HDPO (4:2:1)	$\mathbf{55.0 \pm 1.9}$	$\mathbf{55.0 \pm 1.9}$	$\mathbf{62.0 \pm 1.8}$	$\mathbf{40.0 \pm 1.7}$	$\mathbf{48.0 \pm 1.8}$