DIFFRTL: DIFFERENTIAL REWARD MODELS FOR SYNTHESIS-EFFICIENT REINFORCEMENT LEARNING OF RTL OPTIMIZERS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training-time reinforcement learning is the standard route to endowing open-source language models with register-transfer-level (RTL) design ability, but rewarding power, performance, and area (PPA) requires logic synthesis per design, orders of magnitude slower than any other check in the loop. This synthesis bottleneck caps affordable training, and learned predictors of absolute PPA fall short as substitutes, since typical errors exceed the effect sizes of most optimizations. We observe that optimization consumes only the relative effect of a change, which is far easier to learn because paired comparisons cancel systematic errors. DiffRTL trains a differential reward model on verified transformation pairs manufactured by a paired data engine and uses a fast forward pass of the model as the PPA reward, corrected during training by periodic real-synthesis audits. Every reported number remains grounded in real simulation, formal equivalence checking, and synthesis. Spending 640 synthesis calls on audits alone, DiffRTL beats synthesis-rewarded training at six times the budget by 6.8 eff@1 points on Pluto, with matching gains on RTLLM v2, at a fraction of the tool time. The differential formulation surpasses an absolute predictor trained on identical data. Audits measure reward over-optimization exactly and suppress it throughout training.

1 INTRODUCTION

Modern chips begin as register-transfer-level (RTL) code: engineers describe hardware behavior in languages such as Verilog, and a compilation step called logic synthesis turns the description into a network of gates. The quality of the resulting circuit is summarized by its power, performance, and area (PPA), which determine how much energy the chip draws, how fast it runs, and how much silicon it costs. As integrated circuits grow more complex, manual RTL design has become a costly bottleneck, and large language models (LLMs) are stepping in as capable RTL engineers, with benchmarks such as VerilogEval (Pinckney et al., 2025) and RTLLM (Liu et al., 2024) accelerating progress. However, correct code is not the same as efficient code: first implementations, whether written by engineers or generated by models, usually leave substantial room for PPA improvement. The frontier of the field is therefore shifting from generating functionally correct code to improving PPA. Training-time reinforcement learning (RL) has become the standard route to PPA-aware ability for open-source models (Chen et al., 2026; 2025; Zhu et al., 2026; Zhang et al., 2025). In a typical pipeline, the policy model proposes an RTL implementation, executable tools evaluate the proposal, and the evaluation outcome becomes the reward that updates the policy. Rewarding functional correctness is inexpensive because simulation against a testbench finishes within seconds. Rewarding PPA is fundamentally harder: trusted PPA values come only from logic synthesis, and a single synthesis invocation costs orders of magnitude more time.

Synthesis cost therefore caps how much PPA feedback a training run can afford. ChipSeek-R1 (Chen et al., 2025) trains its RL stage on 8,453 samples after supervised initialization. CodeV-R1 (Zhu et al., 2026) describes the computation cost of RL with verifiable rewards as prohibitive. PPA-RTL (Zhao et al., 2025) constructs synthesis-ranked preference data from only 3,000 sampled descriptions. A tempting shortcut is to replace synthesis with a learned predictor of absolute PPA (Abdollahi et al., 2025; Fang et al., 2023). However, even when 10–20% errors are tolerated, Me-

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.

tRex (Abdelatty et al., 2025a) reports only 43–73% prediction accuracy. The effect of a source-level optimization is often below five percent, so optimization signal drowns in prediction noise.

The difficulties above share two root causes. First, trusted PPA feedback is structurally expensive, because reliable PPA values must be produced by logic synthesis running inside electronic design automation (EDA) tools. An RL run demands thousands of reward queries, and every rollout that clears functional checks costs a full synthesis run under every existing recipe. Second, existing PPA predictors adopt an absolute formulation that mismatches what optimization needs. A predictor of absolute PPA spends its capacity on design-level magnitudes that vary with tools, libraries, and coding style, while the training loop consumes only the relative ordering of candidate changes.

To address the expensive synthesis feedback and the mismatched absolute formulation, we propose DiffRTL, a training-time RL framework that replaces the synthesis reward with a learned differential reward model. The reward model scores a change rather than a design: given a base design and a candidate edit, the model predicts the relative PPA effect of the edit. Sharing the base cancels systematic error, and controlled experiments manufacture paired labels at scale. The same principle let DiffHLS (Peng et al., 2026) cut estimation error roughly fivefold in high-level-synthesis pragma spaces, and PIE (Shypula et al., 2024) showed that edit-level supervision works for software performance. Every rollout that passes cheap correctness checks receives its PPA reward from one fast forward pass of the reward model, so reward throughput no longer bounds training. Periodic audits keep the reward model aligned with real synthesis even as the policy learns to exploit model errors (Gao et al., 2023). Each audit synthesizes a small batch of fresh rollouts, measures the gap between predicted and realized improvement, and updates the reward model. The reward model influences only training, never measurement: every reported number comes from real simulation, formal equivalence checking, and synthesis. An inaccurate reward model can therefore waste training compute at worst but cannot fabricate a result. Spending 640 synthesis calls on audits alone, DiffRTL beats synthesis-rewarded RL at six times the budget by 6.8 eff@1 points on Pluto, with matching gains on RTLLM v2, while pricing the same run with real synthesis would cost 125$\times$ more tool time.

Our contributions are threefold:

- We present DiffRTL, a training-time RL framework for RTL-to-RTL optimization, the task of rewriting a functionally correct design into a PPA-superior equivalent. The framework replaces the synthesis reward with a learned differential reward model and is backed by a paired data engine that manufactures 1.13×10^5 precisely labeled transformation pairs.
- We introduce an audited surrogate-reward recipe for RL, in which periodic real-synthesis audits measure the gap between predicted and realized improvement and correct the reward model during training, and we provide the first exact measurement of reward over-optimization against an executable oracle.
- Experiments on the Pluto, RTLLM v2, and RTL-OPT benchmarks show that DiffRTL beats synthesis-rewarded RL at six times the synthesis budget by 6.8 eff@1 points, at 125$\times$ less reward tool time. The differential formulation surpasses absolute predictors trained on identical data by 16.0 AUC points and transfers across design families.

2 Related Work

2.1 LLM-Based RTL Generation and Optimization

Spec-to-RTL generation established the field through correctness-centered training (Zhu et al., 2026; Zhang et al., 2025). As attention shifts to PPA, ChipSeek (Chen et al., 2026) and ChipSeek-R1 (Chen et al., 2025) integrate EDA toolchains into the training loop and reward synthesis outcomes hierarchically, while PPA-RTL (Zhao et al., 2025) trains on synthesis-ranked preferences, corroborating that relative feedback suits PPA optimization. A parallel line keeps the model frozen and pays synthesis at deployment instead, through evolutionary search (Min et al., 2026; Hsin et al., 2026) and agentic rewriting (Yao et al., 2024; Wang et al., 2026; Chang et al., 2026; Li et al., 2026; Fang et al., 2026), where every candidate is evaluated by real tool runs and the accumulated experience dissolves once the run ends. Test-time training pushes per-design specialization furthest: Alpha-RTL (Zhou et al., 2026) updates the policy on every single design with synthesis-scored rollouts. In all of the above, PPA feedback is purchased one synthesis run at a time, whether during training or for each

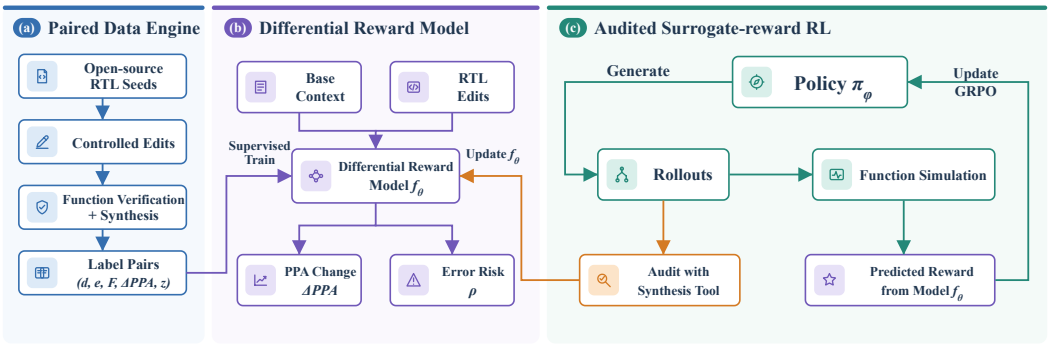


Figure 1: DiffRTL overview. (a) A paired data engine manufactures verified edit pairs. (b) A differential reward model learns to price edits. (c) RL uses the model's forward pass as the PPA reward, corrected by periodic synthesis audits.

deployed design. DiffRTL breaks this coupling: a differential reward model, trained once, supplies PPA feedback in a fraction of a second and is amortized across designs.

2.2 Learned Estimation of Circuit Quality

Learned estimators of circuit quality differ along two axes: what they predict and how the prediction is consumed. Most hardware predictors estimate absolute PPA from RTL, with fine-tuned language models (Abdelatty et al., 2025a; Abdollahi et al., 2025), structural graphs (Fang et al., 2023; Liu et al., 2025; Hemadri et al., 2026), or multimodal encoders (Fang et al., 2025). All bind to a single synthesis flow and technology library, an assumption that MetRex (Abdelatty et al., 2025a) makes explicit. DiffHLS (Peng et al., 2026) moves to relative prediction for pragma changes in high-level synthesis, but its edit space is closed and its outputs remain estimates. Outside hardware, learned cost models rank candidates inside compiler search (Mendis et al., 2019; Chen et al., 2018), edit-level learning captures the effect of code changes (Shypula et al., 2024; Yin et al., 2018; Copet et al., 2025; Yang et al., 2026), and program search consumes executable scores at scale (Novikov et al., 2025; Ouyang et al., 2025). DiffRTL differs on the two axes at once: it predicts the relative effect of open code edits, and the prediction is consumed as a reward that a policy actively optimizes, a pressure that passive estimators never face and that motivates the protections examined next.

2.3 Reward Over-Optimization and Conservative Training

Optimizing a policy against a learned score is central to alignment training (Christiano et al., 2017) and invites over-optimization: Gao et al. (2023) quantify how policy quality degrades smoothly as optimization pressure grows, and Shao et al. (2025) show that even seemingly verifiable signals can mislead RL with verifiable rewards. Existing defenses either reshape the reward toward pessimism by penalizing uncertainty (Coste et al., 2024; Yu et al., 2020; Janner et al., 2019), or remove the explicit reward model altogether, folding it into the policy with direct preference optimization (DPO) (Rafailov et al., 2023) or consuming only group-relative comparisons with group relative policy optimization (GRPO) (Shao et al., 2024). Defenses of either kind act on the shape of the reward or the form of the algorithm, because the true objective in human-preference settings cannot be queried. DiffRTL faces the same pressure with one added power: its objective is executable. Periodic audits therefore query real synthesis, measure the gap between predicted and realized improvement, and repair the reward model exactly where the policy exploits it, turning over-optimization from a feared quantity into a measured one.

3 Problem Formulation

An instance of RTL-to-RTL optimization consists of a functionally correct design d_0, a testbench Φ, and objective weights $\lambda = (\lambda_m)_{m \in \mathcal{M}}$ over the metric set $\mathcal{M} = \{\text{area}, \text{delay}, \text{power}\}$. A synthesis flow F, meaning a synthesis tool together with a technology library and a constraint set, maps a

design d to its measured quality $y(d) = (y_m(d))_{m\in\mathcal{M}}$, and absolute PPA is defined only with respect to F. A rewrite d' is admissible if it preserves the module interface of d_0, and equivalent, written $d' \equiv d_0$, if it passes simulation against Φ and formal equivalence checking. The task is to find

$$d^\star = \arg\max_{d'\equiv d_0} J_\lambda(d'), \qquad J_\lambda(d') = \sum_{m\in\mathcal{M}} \lambda_m \log \frac{y_m(d_0)}{y_m(d')}, \tag{1}$$

where the logarithmic ratios reward proportional reductions symmetrically across metrics. A policy π_ϕ addresses the task by sampling rewrites $d' \sim \pi_\phi(\cdot \mid d_0, \lambda)$.

For an ordered pair of designs, define the differential effect of moving from d to d' as

$$\Delta_m(d \to d') = \log \frac{y_m(d')}{y_m(d)}, \qquad m \in \mathcal{M}, \tag{2}$$

The identity $J_\lambda(d') = -\sum_m \lambda_m \Delta_m(d_0 \to d')$, which follows directly from Equations (1) and (2), makes precise what the introduction argued in prose: the objective consumes no absolute PPA, only the differential effects of rewrites, and all rewrites within one instance are compared against the shared base d_0. Learning Δ directly is therefore the natural formulation, and the next section builds the framework around it.

Cost is counted in synthesis invocations under three ledgers. The corpus cost C_0 is paid once to label training pairs and is amortized over every policy trained afterwards. The training budget B_{train} counts synthesis calls consumed by one policy-training run. The evaluation cost b_{eval} counts synthesis calls spent on each deployed task and is fixed by the common evaluation protocol rather than by any method. Methods are compared at matched B_{train} under one identical protocol, and every reported quantity comes from the real tools defined above.

4 DIFFRTL

DiffRTL runs in four stages. A paired data engine edits seed designs, verifies every pair, and labels both sides by synthesis under a shared recipe. The engine runs once at corpus cost C_0. A differential reward model f_θ then learns from the labeled pairs to price a candidate edit: one fast forward pass predicts the differential effect of the edit and the risk that the edit breaks equivalence. The policy π_ϕ trains with GRPO, with f_θ supplying the PPA component of the reward in place of synthesis. Periodic audits correct f_θ during training: each audit synthesizes a batch of fresh rollouts, measures the gap between predicted and realized improvement, and updates the model within the training budget B_{train}. The trained policy is finally evaluated by sampling candidates and measuring every one of them with the real flow. Figure 1 and Algorithm 2 in Appendix D summarize the pipeline, and the following subsections walk through the stages in order.

4.1 PAIRED DATA ENGINE

The engine turns a corpus of seed designs into labeled sibling groups. An edit e transforms a base design d into a rewrite $e(d)$, and a sibling group $G(d) = \{e_1, \ldots, e_K\}$ collects the labeled edits of one base d. Seeds are collected from public RTL corpora, filtered to synthesizable modules, and deduplicated against every evaluation benchmark, so that no test design or near-duplicate leaks into training. Three generators propose edits on each seed: an LLM applies semantic transformations drawn from a taxonomy of hardware optimization patterns, rule-based rewriters produce equivalence-preserving edits that are correct by construction, and a corruption generator injects realistic bugs that provide negative correctness labels.

Every candidate is then checked for equivalence: the rewrite must preserve the interface of the seed, pass functional simulation under the seed testbench, and pass formal equivalence checking on a sampled subset. A failed check is recorded as a correctness label $z(e) = 1$ rather than discarded as waste, and $z(e) = 0$ marks an edit that survives every check. Surviving pairs are labeled by synthesis under a shared recipe: flow, library, constraints, and target period are fixed once per seed and reused for every sibling, so the only varying factor within a group is the edit itself, which makes each group a controlled experiment and each Δ label exact. The base design is synthesized once per seed, and the result is shared by all of the siblings. Repeated synthesis of a sampled subset calibrates

a per-metric noise floor ε_m, below which two effects are indistinguishable. The engine emits the dataset $\mathcal{D} = \{(d, e, \Delta(e), z(e))\}$.

4.2 Differential Reward Model

The second stage trains the reward model on $\mathcal{D}$. A query (d, e) is encoded as two inputs: a context slice of the base design and the edit as a unified diff. The slice covers the edited region plus a bounded fan-in and fan-out neighborhood on the dataflow graph, which decouples input length from design size and forces the model to learn local edit patterns that recur across designs.

The model is a mapping $f_\theta : (d, e) \mapsto \big((\hat{\Delta}_m(e))_{m\in\mathcal{M}},\ \hat{\rho}(e)\big)$, predicting the effect of Equation (2) for each metric together with an equivalence risk $\hat{\rho}(e)$, the probability that the edit breaks equivalence. Supervision is organized by sibling group, so that comparisons inside a group cancel design-level idiosyncrasies by construction, the error cancellation argued in the introduction. A regression term anchors $\hat{\Delta}_m$ to measured effects, and a listwise Plackett–Luce term (Plackett, 1975; Xia et al., 2008) orders each group of K edits by realized effect,

$$\mathcal{L}_{\text{rank}} = -\sum_{m\in\mathcal{M}} \sum_{i=1}^{K} \log \frac{\exp\big(-\hat{\Delta}_m(e_{\sigma_m(i)})\big)}{\sum_{j=i}^{K} \exp\big(-\hat{\Delta}_m(e_{\sigma_m(j)})\big)}, \tag{3}$$

where σ_m sorts the group from most to least improving on metric m, sharpening exactly the comparisons that every consumer of the scores makes. Edits whose effects differ by less than the noise floor ε_m are treated as ties. The risk head classifies the correctness label, and the total loss combines the terms,

$$\mathcal{L}(\theta) = \mathcal{L}_{\text{reg}} + \mu_r\, \mathcal{L}_{\text{rank}} + \mu_z\, \mathcal{L}_{\text{risk}}, \tag{4}$$

where $\mathcal{L}_{\text{reg}}$ sums Huber losses $\ell_{\text{h}}\big(\hat{\Delta}_m(e) - \Delta_m(e)\big)$ over all labeled edits and metrics, $\mathcal{L}_{\text{risk}}$ is the cross entropy of $\hat{\rho}(e)$ against $z(e)$, and μ_r, μ_z are weights. The sibling group recurs at every consumption point. GRPO normalizes advantages within a group of rollouts on one instance, and evaluation compares the candidates of one design against each other. The training unit therefore coincides with every decision unit the scores will serve. Every query stays one hop from a real design, so errors never compound through chains of predictions, and $\hat{\rho}$ is a pre-filter only, with equivalence always decided by real verification. The trained reward model prices an edit in a fraction of a second. The third stage places the model inside the training loop.

4.3 Reward-Model RL with Verifiable Audits

Policy training starts from a supervised warm start: π_ϕ is fine-tuned on verified improving edits from the engine. The warm start teaches the output format and a first layer of competence, and the penalty tier therefore fires rarely once GRPO begins to optimize the policy. For each training instance (d, λ) the policy samples a group of G rewrites, and each rewrite is scored by a hierarchical reward with the structure shared by synthesis-rewarded RL (Chen et al., 2025; Zhou et al., 2026): functional simulation gates correctness, and f_θ prices only the tier that used to dominate evaluation cost,

$$r(e \mid d, \lambda) = \begin{cases} r_{\text{fail}} & \text{simulation fails,} \\ -\sum_m \lambda_m\, \hat{\Delta}_m(e) & \text{otherwise,} \end{cases} \tag{5}$$

where $r_{\text{fail}} < 0$ is a fixed penalty, and a malformed or uncompilable rewrite fails simulation trivially. Functional simulation runs per rollout and finishes in seconds. The last tier is the predicted objective $\hat{J}_\lambda(e) = -\sum_m \lambda_m \hat{\Delta}_m(e)$ of Equation (1), and the advantage of rollout i is normalized within the group of G rollouts,

$$\hat{A}_i = \frac{r_i - \text{mean}(r_{1:G})}{\text{std}(r_{1:G})}, \tag{6}$$

the same within-group comparison the ranking loss was trained on. Pricing the PPA tier takes one fast forward pass instead of one synthesis run, so the number of policy updates decouples from the synthesis budget.

Audits are the sole defense against over-optimization and the only consumer of B_{train}. Every N policy steps, an audit draws A fresh rollouts, synthesizes and verifies them with the real flow, and

Table 1: Main results. Mean±std over three seeds. Best in bold. All trained rows share the base model and the evaluation protocol. RL time is the total wall clock of one RL run, including synthesis, on eight A100s with 32 parallel synthesis workers. Search-based baselines omit Pluto, whose eff@1 is defined over sampling rather than search.

Method	Pluto eff@1 (%)	RTLLM PPA impr. (%)	RTL-OPT impr./36	B_{train}	RL time (h)	synth./task
Frozen base	36.2±1.1	4.8±1.6	8	0	—	8.4
SFT only	43.2±1.2	10.9±1.9	14	0	—	9.3
Synthesis@640	43.9±1.4	12.1±2.0	15	640	0.7	9.3
Synthesis@2000	45.9±1.6	14.1±2.1	17	2,000	2.1	9.3
Synthesis@4000	46.8±1.7	15.4±2.1	18	4,000	4.4	9.3
DiffRTL (ours)	**53.6**±1.4	**27.4**±1.8	**26**	640	4.5	9.6
+ RM rerank	53.1±1.4	26.3±1.8	24	640	4.5	**3.8**
Frozen-model per-design search (deployment-side synthesis)						
REvolution (repro.)	—	24.8±2.3	23	—	—	287/design
EvolVE (repro.)	—	25.1±2.1	24	—	—	≈240/design

measures the audit gap

$$g = \frac{1}{A} \sum_{i=1}^{A} \left[\hat{J}_\lambda(e_i) - J_\lambda(d_i') \right], \tag{7}$$

where d_i' is the rewrite that e_i produces and J_λ is measured by the real flow. The audit serves two purposes. Audit labels update f_θ, repairing the model exactly where the policy exploits errors. The gap trace across audits is reported as the measurement of over-optimization. If g exceeds a threshold τ_g on consecutive audits, the schedule hardens: the audit interval halves and the KL anchor to the warm start strengthens. The division of labor is strict: f_θ decides only which rollouts are reinforced, while equivalence and PPA of anything reported are measured by the real tools.

4.4 Deployment and Evaluation

A deployed instance (d_0, Φ, λ) is served by the standard protocol of the field (Abdelatty et al., 2025b; Zhu et al., 2026). The policy samples k candidate rewrites. Every candidate is simulated against Φ, the survivors are synthesized and formally verified, and the best verified rewrite is returned with measured PPA. Every method compared in the experiments is served by this identical protocol, so differences in results reflect training alone. The reward model takes no part in the protocol. The experiments separately examine reranking with the reward model, which cuts how many candidates need synthesis.

5 Experiments

Our experiments answer five research questions. **RQ1**: Can the learned differential reward replace synthesis inside the training loop at a matched budget? **RQ2**: How much training efficiency and deployment cost does the replacement buy? **RQ3**: Does the differential formulation beat absolute prediction? **RQ4**: Can audits measure reward over-optimization exactly, and does correcting the reward model suppress it? **RQ5**: How much does each core component of DiffRTL contribute to the final quality?

5.1 Experimental Setup

Benchmarks. We evaluate on three PPA-optimization benchmarks. Pluto (Abdelatty et al., 2025b) contains 114 optimization problems. RTLLM v2 (Liu et al., 2024) contains 50 designs. RTL-OPT (Lu et al., 2026) contains 36 optimization pairs. Each task provides a testbench and a reference implementation, which serves as the base design d_0. The Pluto reference is expert-optimized and defines the efficiency bar.

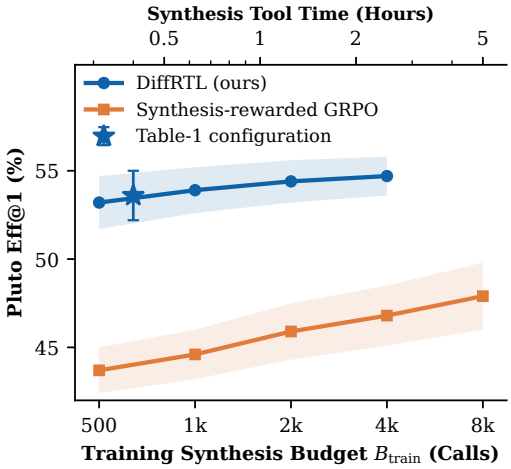


Figure 2: Pluto eff@1 against the training synthesis budget. The star marks the Table-1 configuration. All points use three seeds, and bands show one standard deviation.

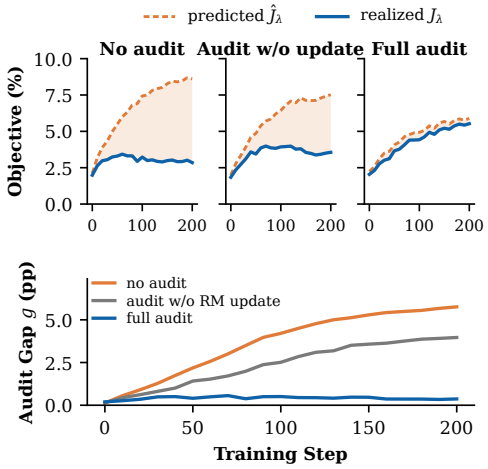


Figure 3: Reward over-optimization under three audit protocols: predicted versus realized objective (top) and the audit gap of Equation (7) (bottom). Probes run every 10 steps.

Metrics. Pluto reports eff@1 $= \frac{1}{|T|}\sum_{t\in T} c_t/n$, where c_t counts, among the n=16 samples for task t, the rewrites that are functionally correct and at least as efficient as the reference. RTLLM v2 reports the geometric-mean change of the PPA product over the 49 synthesizable designs, and the remaining design is checked by simulation only. RTL-OPT reports the number of improved designs. Reward-model tests report pairwise ranking AUC. Every Table-1 configuration and every point of Figure 2 uses three seeds, whose results are reported as mean and standard deviation.

Baselines. For the training-paradigm comparison of RQ1 and RQ2, five settings share the base policy model and the evaluation protocol: (1) the frozen base, (2) SFT only, (3) synthesis-rewarded GRPO in the ChipSeek-R1 recipe (Chen et al., 2025), reported as Synthesis@B, where $B \in \{640, 2{,}000, 4{,}000\}$ is the synthesis-call budget spent on rewards during RL, (4) DiffRTL, and (5) DiffRTL with RM rerank, which keeps the trained policy and changes only evaluation, where f_θ scores the sampled candidates and only the top-ranked few are synthesized for final selection (RQ2). For quality context on RTLLM v2, REvolution (Min et al., 2026) and EvolVE (Hsin et al., 2026) are reproduced as per-design search over frozen models, with per-design synthesis cost annotated. For the formulation comparison of RQ3, four reward models share the Qwen2.5-Coder-3B backbone: (1) Absolute, scoring an edit by the predicted PPA of the edited design alone, (2) Absolute+diff, scoring by the difference of two such predictions at test time, (3) Differential w/o rank, trained with the regression term of Equation (4) only, and (4) Differential, the full model of the method section. For the audit comparison of RQ4, three protocols are trained: no audit, audit without reward-model update, and the full protocol.

Implementation. All synthesis and PPA analysis use one open flow: Yosys for synthesis and area, and OpenSTA for timing and power. The flow follows the practice of synthesis-rewarded RL (Chen et al., 2025; Zhou et al., 2026). Functional checks use Icarus Verilog simulation and SymbiYosys equivalence checking. All experiments run on eight NVIDIA A100 GPUs. Under this flow, the paired data engine of our DiffRTL turns 18,400 seed designs, synthesizable Verilog modules collected from open-source corpora, into 1.13×10^5 labeled pairs. Exact or near duplicates of any test design are removed from the training corpus. The corpus cost is $C_0 = 1.33 \times 10^5$ syntheses under Nangate45, about 2.7×10^3 CPU-hours, or roughly 3.5 days across 32 parallel synthesis workers. The reward model f_θ is a Qwen2.5-Coder-3B backbone with two linear heads that output $\hat{\Delta}$ and $\hat{\rho}$, adapted with LoRA and trained on the corpus by Equation (4). The policy model π_ϕ is Qwen2.5-Coder-7B, also adapted with LoRA. The policy is warm-started by SFT on 3×10^4 formally verified improving edits and then trained with GRPO for 200 steps, about 0.7 epochs over the instance pool, with 64 instances and G=8 rollouts per step under a KL anchor. Audits run every N=10 policy

Table 2: Ranking AUC of the four reward-model formulations. RTLOpt reports pairwise accuracy.

Formulation	in-fam.	x-fam.	RTLOpt
Absolute	0.664	0.612	0.541
Absolute+diff	0.701	0.648	0.562
Differential w/o rank	0.793	0.752	0.671
Differential (ours)	**0.824**	**0.781**	**0.703**

steps on A=32 fresh rollouts, consume 640 synthesis calls in total, and trigger hardening when the gap of Equation (7) stays above τ_g=3 percentage points.

5.2 Main Results (RQ1)

Table 1 reports the main comparison against the baselines. The Synthesis settings follow the ChipSeek-R1 recipe (Chen et al., 2025) and scale the training synthesis budget. More budget buys the baseline little: the RTLLM improvement grows from 12.1 to 15.4 points across 640 to 4,000 calls, and the runtime column grows in step, from 0.7 to 4.4 hours, because every rollout pays real synthesis. DiffRTL spends 640 calls on audits alone and beats Synthesis@4000 on every metric: 53.6 against 46.8 eff@1 on Pluto, 27.4% against 15.4% on RTLLM v2, and 26 against 18 improved RTL-OPT designs. At nearly equal runtime, 4.5 against 4.4 hours, DiffRTL holds a 6.8-point eff@1 lead over Synthesis@4000. Per-design search offers a different route to quality and spends synthesis on every design at deployment. Reproduced under our protocol, REvolution and EvolVE reach 24.8% and 25.1% on RTLLM v2 at about 287 and 240 calls per design, while the two DiffRTL configurations reach 27.4% and 26.3% at 9.6 calls per task. Taken together, the results trace one asymmetry. A synthesis call spent as a reward or on a single design is consumed once, while the same call spent auditing the differential model keeps pricing every later rollout, which is how 640 calls end up beating budgets six times larger.

5.3 Training Efficiency and Cost (RQ2)

Efficiency comes down to three prices: one reward, one training run, and one deployed task. One synthesis call takes 1.2 minutes on average on one synthesis worker. Producing one predicted reward takes 0.31 s on one A100, more than two orders of magnitude less. A training run inherits the same asymmetry. One DiffRTL run issues about 8×10^4 predicted rewards, which cost about 0.9 hours on the same eight A100s inside the 4.5-hour total of Table 1. Pricing the same 8×10^4 rewards with real synthesis instead would take 2.1 days of tool time across 32 parallel synthesis workers, against 0.4 hours for the audits, a 125$\times$ gap. Figure 2 sweeps the audit batch on our side and matched budget levels on the baseline side. The upper axis reports the corresponding wall-clock hours across the same 32 parallel synthesis workers. DiffRTL at 500 synthesis calls already exceeds the baseline at 8,000, at least 16$\times$ fewer calls at equal quality. At deployment, f_θ scores the sampled candidates and only the top-ranked few are synthesized. The reranking cuts per-task synthesis from 9.6 to 3.8 at a 0.5-point quality cost (Table 1). A reward learned once, and audited while it works, discounts every scale of the pipeline: the predicted reward that replaces a synthesis call, the training run that spends tool time only on audits, and the deployed task that synthesizes only top-ranked candidates.

5.4 Reward Model Quality (RQ3)

Table 2 validates the reward model itself through four formulation configurations, with absolute prediction as the main contrast. The task is pairwise ranking of verified edits by measured PPA change. Test pairs span three splits: unseen designs in familiar families, unseen families, and the external RTLOpt set of 120 triples released with CODMAS (Chang et al., 2026), which serves as the out-of-distribution test. Differential reaches 0.824 in-family AUC against 0.664 for Absolute, and a 16.0-point gap on shared backbone and training pairs can only come from the formulation. Absolute+diff recovers only part of the gap (0.701), because subtraction cancels shared bias but not the two independent prediction errors. The ranking loss contributes a further 3.1 points over Differential w/o rank. The case for necessity is made off distribution: Differential holds 0.781 across families, while Absolute falls to 0.612. RTLOpt repeats the pattern out of distribution (0.703 against

Table 3: Ablations on Pluto eff@1, mean over three seeds and sorted by degradation. Δ is the drop from the full configuration.

Setting	eff@1 (%)	Δ
Full DiffRTL	53.6	—
no functional simulation gate ($\hat{\rho}$ decides)	33.5	-20.1
no audit	47.2	-6.4
RM data 10K	48.1	-5.5
audit w/o RM update	49.0	-4.6
RM data 30K	51.2	-2.4
w/o ranking loss (μ_r=0)	51.5	-2.1
audit interval N=40	51.9	-1.7

0.541). The differential formulation therefore beats absolute prediction on every split, and the collapse of Absolute off distribution makes the differential design a necessity rather than a refinement.

5.5 Over-Optimization Analysis (RQ4)

Figure 3 turns the audit gap of Equation (7) into a measurement, tracing predicted $\hat{J}_\lambda$ against realized J_λ under three protocols with no component ablated. Probes of A=32 fresh rollouts run every 10 steps from the shared warm start. The no-audit arm delivers the measurement the introduction called for: predicted improvement climbs to 8.7% while realized quality peaks near step 68 and then decays, an exact 5.7 pp account of over-optimization against the tool itself. The policy optimizes the predictor, not the design. Auditing without updating f_θ slows the drift, and the one hardening event near step 136 marks the visible kink, yet the gap still reaches 4.0 pp: an alarm slows what only repair can stop. The full protocol keeps the gap below 1 pp and never triggers hardening. One audit removes about two thirds of the accumulated drift (0.45 to 0.15 pp). Throughout, 78% of training rollouts pass simulation, so the penalty tier stays rare and audits remain the only synthesis consumer. The three protocols end at 47.2, 49.0, and 53.6 eff@1: measured exactly, over-optimization is also suppressed, and the budget that measures it is the budget that repairs it.

5.6 Ablations (RQ5)

Table 3 measures the contribution of every component by the final Pluto eff@1. The largest degradation validates the hierarchical reward: when $\hat{\rho}$ replaces the real functional simulation gate, correctness collapses (candidate pass rate 52% to 31%) and eff@1 falls to 33.5, below the frozen base. A predicted gate invites the broken code that a real gate rejects. The audit family repeats the dynamics of Figure 3 in final quality: removing audits costs 6.4 points, restoring measurement and hardening alone recovers 1.8 of them, and updating the model recovers the rest. Reward-model data scales with diminishing but unsaturated returns, 48.1 to 51.2 to 53.6 across 10K, 30K, and 113K pairs, and the matching AUC chain (0.741, 0.789, 0.824) ties policy quality to ranking quality. The two smallest effects, removing the ranking loss (-2.1) and sparsening audits to N=40 (-1.7), are directionally consistent but within one standard deviation. Every component earns a positive share, and the two that touch reality, the functional simulation gate and the audits, earn the largest.

6 Conclusion

DiffRTL replaces the synthesis reward in RTL optimization RL with a differential model that prices the effect of an edit, and corrects the model through synthesis audits. A paired data engine manufactures the verified transformation pairs that make this reward learnable at scale. Under 640 audit calls, the policy beats Synthesis@4000, the baseline at six times the budget, by 6.8 eff@1 points. The differential formulation proves necessary rather than optional and transfers across design families, and the audits turn over-optimization from a risk into a measured and repaired quantity. At deployment, the same model reranks sampled candidates and cuts per-task synthesis further. A reward learned once, and audited while it works, makes PPA-level reinforcement learning affordable.

AI USE STATEMENT

In this work, we used generative AI tools for limited language editing, paraphrasing, and polishing of the manuscript text. We have not used generative AI tools for research ideation, experimental design, implementation, data analysis, or the generation of any technical content, claims, or results. We have reviewed all AI-assisted work: all AI-edited text was checked and verified by the authors, and all technical content, analysis, experimental results, and conclusions were developed and verified by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personally identifiable information. Seed designs come from public open-source RTL corpora, and all benchmarks are publicly available and used under their respective licenses. DiffRTL inherits the failure modes of its tools and learned components. The reward model is trained under one open synthesis flow and technology library, so its rankings may not transfer to other flows. Training rewards gate correctness by simulation alone, and weak testbenches can therefore admit incorrect edits during training. Every reported rewrite passes simulation and formal equivalence checking against its base design, yet deployment in production hardware would still require sign-off with the target toolchain. Beyond the one-time compute of corpus construction, about 2.7×10^3 CPU-hours of synthesis, we do not foresee direct negative societal impacts specific to this work.

REPRODUCIBILITY STATEMENT

Our code is available in an anonymous repository at `https://anonymous.4open.science/r/DiffRTL-53AF/README.md`. Section 5.1 describes the benchmarks and metrics, the baselines, and the synthesis flow and model configurations. Appendix B specifies the paired data engine, including (1) seed curation and leak protection against the test benchmarks, (2) the edit generators and optimization taxonomy, (3) the verification cascade, (4) the synthesis recipe and PPA measurement, and (5) noise-floor calibration. Appendices C and D detail the reward model and the audited RL loop, with all hyperparameters listed in Tables 11 and 12. Appendix E defines the evaluation protocol and every metric, Appendix F documents the baselines, and Appendix G derives every reported cost figure. Appendix I covers software, hardware, and determinism, and Table 17 maps each pipeline stage to a command of the released implementation. Every tool in the flow, the Nangate45 library, and the Qwen2.5-Coder backbones are publicly available.

REFERENCES

M. Abdelatty, J. Ma, and S. Reda. MetRex: A benchmark for Verilog code metric reasoning using LLMs. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, pp. 995–1001, 2025a.

M. Abdelatty, M. Nouh, J.K. Rosenstein, and S. Reda. Pluto: A benchmark for evaluating efficiency of LLM-generated hardware code. *arXiv preprint arXiv:2510.14756*, 2025b.

A. Abdollahi, M. Kamal, and M. Pedram. RocketPPA: Ultra-fast LLM-based PPA estimator at code-level abstraction. *arXiv preprint arXiv:2503.13750*, 2025.

C.M. Chang, P. Vijayaraghavan, A. Jadhav, C. Mackin, H. Tsai, V. Mukherjee, and E. Degan. CODMAS: A dialectic multi-agent collaborative framework for structured RTL optimization. In *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 5: Industry Track)*, pp. 777–788, 2026.

T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

Z. Chen, K. Chang, Z. Li, X. He, C. Chen, C. Li, M. Wang, H. Xu, Y. Han, and Y. Wang. ChipSeek-R1: Generating human-surpassing RTL with LLM via hierarchical reward-driven reinforcement learning. *arXiv preprint arXiv:2507.04736*, 2025.

Z. Chen, K. Chang, Z. Li, C. Li, X. He, C. Chen, M. Wang, H. Xu, Y. Han, H. Li, and Y. Wang. ChipSeek: Optimizing Verilog generation via EDA-integrated reinforcement learning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25180–25201, 2026.

P.F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30, 2017.

J. Copet, Q. Carbonneaux, G. Cohen, J. Gehring, J. Kahn, J. Kossen, F. Kreuk, E. McMilin, M. Meyer, Y. Wei, and D. Zhang. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

T. Coste, U. Anwar, R. Kirk, and D. Krueger. Reward model ensembles help mitigate overoptimization. In *International Conference on Learning Representations*, pp. 50905–50931, 2024.

W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L.W. Wills, H. Zhang, and Z. Xie. MasterRTL: A pre-synthesis PPA estimation framework for any RTL design. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9. IEEE, 2023.

W. Fang, S. Liu, J. Wang, and Z. Xie. CircuitFusion: Multimodal circuit representation learning for agile chip design. *arXiv preprint arXiv:2505.02168*, 2025.

W. Fang, Y. Lu, S. Liu, J. Wang, Z. Guo, J. He, F. Tu, and Z. Xie. Dr. RTL: Autonomous agentic RTL optimization through tool-grounded self-improvement. *arXiv preprint arXiv:2604.14989*, 2026.

L. Gao, J. Schulman, and J. Hilton. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pp. 10835–10866. PMLR, 2023.

R.V. Hemadri, J. Bhandari, A. Nakkab, J. Knechtel, B. Gopalan, R. Narayanaswamy, R. Karri, and S. Garg. VeriLoC: Line-of-code level prediction of hardware design quality from Verilog code. *Advances in Neural Information Processing Systems*, 38:13541–13568, 2026.

W.P. Hsin, R.H. Deng, Y.T. Hsieh, E.M. Huang, and S.H. Hung. EvolVE: Evolutionary search for LLM-based Verilog generation and optimization. *arXiv preprint arXiv:2601.18067*, 2026.

M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

C. Li, P. Jin, D. Huang, G. Sun, H. Han, J. Mu, X. Zheng, J. Zhu, S. Xing, H. Wei, and T. Ma. AutoPPA: Automated circuit PPA optimization via contrastive code-based rule library learning. *arXiv preprint arXiv:2604.18445*, 2026.

S. Liu, Y. Lu, W. Fang, M. Li, and Z. Xie. OpenLLM-RTL: Open dataset and benchmark for LLM-aided RTL design. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024.

Y. Liu, H. Zhang, Y. Wang, D. Tsaras, L. Chen, M. Yuan, and Q. Xu. Beyond tokens: Enhancing RTL quality estimation via structural graph learning. *arXiv preprint arXiv:2508.18730*, 2025.

Y. Lu, S. Liu, H. Zhou, W. Fang, Q. Zhang, and Z. Xie. A new benchmark for the appropriate evaluation of RTL code optimization. *arXiv preprint arXiv:2601.01765*, 2026.

C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pp. 4505–4515. PMLR, 2019.

K. Min, K. Cho, J. Jang, and S. Kang. REvolution: An evolutionary framework for RTL generation driven by large language models. In *2026 31st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 282–288. IEEE, 2026.

A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.S. Huang, A.Z. Wagner, S. Shirobokov, B. Kozlovskii, F.J. Ruiz, A. Mehrabian, and M.P. Kumar. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

A. Ouyang, S. Guo, S. Arora, A.L. Zhang, W. Hu, C. Ré, and A. Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Z. Peng, Z. Li, Q. Xu, and J. Zhao. DiffHLS: Differential learning for high-level synthesis QoR prediction with GNNs and LLM code embeddings. *arXiv preprint arXiv:2604.09240*, 2026.

N. Pinckney, C. Batten, M. Liu, H. Ren, and B. Khailany. Revisiting VerilogEval: A year of improvements in large language models for hardware code generation. *ACM Transactions on Design Automation of Electronic Systems*, 30(6):1–20, 2025.

R.L. Plackett. The analysis of permutations. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 24(2):193–202, 1975.

R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C.D. Manning, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

R. Shao, S.S. Li, R. Xin, S. Geng, Y. Wang, S. Oh, S.S. Du, N. Lambert, S. Min, R. Krishna, and Y. Tsvetkov. Spurious rewards: Rethinking training signals in RLVR. *arXiv preprint arXiv:2506.10947*, 2025.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y.K. Li, Y. Wu, and D. Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

A.G. Shypula, A. Madaan, Y. Zeng, U. Alon, J.R. Gardner, Y. Yang, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024.

Y. Wang, W. Ye, P. Guo, Y. He, Z. Wang, B. Tian, S. He, G. Sun, Z. Shen, S. Chen, and A. Srivastava. SymRTLO: Enhancing RTL code optimization with LLMs and neuron-inspired symbolic reasoning. *Advances in Neural Information Processing Systems*, 38:50093–50118, 2026.

F. Xia, T.Y. Liu, J. Wang, W. Zhang, and H. Li. Listwise approach to learning to rank: Theory and algorithm. In *Proceedings of the 25th International Conference on Machine Learning*, pp. 1192–1199, 2008.

J. Yang, W. Zhang, J. Wu, J. Cheng, T. Zheng, F. Xu, W. Gu, L. Jing, Y. Du, J. Li, and Y. Li. InCoder-32B-Thinking: Industrial code world model for thinking. *arXiv preprint arXiv:2604.03144*, 2026.

X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu. RTLRewriter: Methodologies for large models aided RTL code optimization. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–7, 2024.

P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A.L. Gaunt. Learning to represent edits. *arXiv preprint arXiv:1810.13337*, 2018.

T. Yu, G. Thomas, L. Yu, S. Ermon, J.Y. Zou, S. Levine, C. Finn, and T. Ma. MOPO: Model-based offline policy optimization. *Advances in Neural Information Processing Systems*, 33:14129–14142, 2020.

Y. Zhang, R. Zhang, J. Guo, L. Huang, D. Huang, Y. Zhao, S. Cheng, P. Jin, C. Li, Z. Du, and X. Hu. QiMeng-SALV: Signal-aware learning for Verilog code generation. *arXiv preprint arXiv:2510.19296*, 2025.

Y. Zhao, W. Fu, S. Li, Y.X. Hu, X. Guo, and Y. Jin. Hardware generation with high flexibility using reinforcement learning enhanced LLMs. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pp. 1–7. IEEE, 2025.

P. Zhou, Z. Chen, C. Li, H. Gao, K. Chang, Z. Qu, and Y. Wang. Alpha-RTL: Test-time training for RTL hardware optimization. *arXiv preprint arXiv:2606.05253*, 2026.

Y. Zhu, D. Huang, H. Lyu, X. Zhang, C. Li, W. Shi, Y. Wu, J. Mu, J. Wang, P. Jin, and S. Cheng. QiMeng-CodeV-R1: Reasoning-enhanced Verilog generation. *Advances in Neural Information Processing Systems*, 38:154266–154300, 2026.

Table 4: Notation.

Symbol	Meaning
d_0, d	functionally correct base design
Φ	testbench of the instance
λ	objective weights on M, $\sum_m \lambda_m = 1$
F	synthesis flow: tool + library + constraints
$y(d)$	measured quality $(y_m(d))_{m\in M}$ under F
e, $e(d)$	an edit and the design it produces
$G(d)$	sibling group $\{e_1, \ldots, e_K\}$ of one base d
$\Delta_m(e)$	differential effect $\log\big(y_m(e(d))/y_m(d)\big)$
J_λ	objective of Eq. (1) of the main paper
$z(e)$	correctness label; 1 iff the edit breaks equivalence
$\hat{\rho}(e)$	predicted equivalence risk
ε_m	per-metric noise floor
f_θ	differential reward model
π_ϕ	policy
G	rollouts per instance per step
N, A	audit interval and audit batch
τ_g	audit-gap hardening threshold
C_0	corpus cost, in synthesis invocations
B_{train}	training budget, in synthesis invocations
b_{eval}	per-task evaluation cost, in synthesis invocations

A Notation, Objective, and Cost Model

A.1 Notation

Table 4 collects the symbols used in the main paper and in this appendix. The metric set is fixed to $M = \{\text{area}, \text{delay}, \text{power}\}$ throughout, and every tensor with a metric axis uses that order.

A.2 Why the objective consumes no absolute PPA

The identity that motivates the whole framework follows in one line. Writing Eq. (1) of the main paper and substituting Eq. (2),

$$\begin{aligned} J_\lambda(d') &= \sum_{m\in M} \lambda_m \log \frac{y_m(d_0)}{y_m(d')} \\ &= -\sum_{m\in M} \lambda_m \log \frac{y_m(d')}{y_m(d_0)} \\ &= -\sum_{m\in M} \lambda_m \Delta_m(d_0 \to d'). \end{aligned} \tag{8}$$

Every design-level magnitude $y_m(d_0)$ has cancelled. Two consequences are used repeatedly:

- **The reward needs no absolute prediction.** A model that predicts Δ suffices to evaluate J_λ exactly, so the capacity an absolute predictor spends on magnitudes that vary with tool, library and coding style is capacity the task never asks for.
- **Within a sibling group, differences in Δ and differences in $\log y$ coincide.** Because $\log y_m(e_i(d)) - \log y_m(e_j(d)) = \Delta_m(e_i) - \Delta_m(e_j)$ for any two siblings of the same base, the same noise floor ε_m (Appendix B.5) resolves ties for both the differential and the absolute formulation. This is what allows Table 2 of the main paper to be a comparison of *formulations* under one identical training objective rather than a comparison of loss functions; the contrasts are specified in Appendix C.6.

Positive J_λ always means improvement. We report J_λ in log units internally and multiply by 100 when quoting percentages, so a 1 percentage-point (pp) figure is a 1% relative change to first order.

Algorithm 1 Paired data engine (one seed)

Require: seed d, testbench Φ, flow F, generators $\mathcal{G}$
$R \leftarrow \textsc{FixRecipe}(d, F)$ {library, constraints, period}
$y(d) \leftarrow \textsc{Synthesize}(d, R)$ {once; shared by the group}
if $y(d)$ is unavailable **then**
  return $\emptyset$ {seed dropped, no siblings labeled}
end if
$E \leftarrow \bigcup_{g \in \mathcal{G}} g.\textsc{Propose}(d, n_g)$
$\mathcal{D} \leftarrow \emptyset$
for $e \in E$ (deduplicated, capped at K) **do**
  $v \leftarrow \textsc{Verify}(d, e, \Phi)$ {interface $\rightarrow$ sim $\rightarrow$ sampled formal}
  if v failed **then**
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(d, e, \perp, z{=}1)\}$ {kept, not discarded}
  else
    $y(e(d)) \leftarrow \textsc{Synthesize}(e(d), R)$
    if $y(e(d))$ is unavailable **then**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(d, e, \perp, z{=}1)\}$
    else
      $\Delta(e) \leftarrow \log\big(y(e(d))/y(d)\big)$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(d, e, \Delta(e), z{=}0)\}$
    end if
  end if
end for
return $\mathcal{D}$

A.3 Sign and weight conventions

With uniform λ, J_λ is the negative log of the geometric mean of the three PPA ratios, so $\exp(-3J_\lambda)$ is exactly the PPA-product ratio that the RTLLM v2 metric measures. All main-paper experiments use uniform $\lambda = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. The implementation also supports per-instance Dirichlet sampling of λ (configuration key `rl.lam_sampling`), which conditions the policy on the weight vector through the prompt; this option is not used for any reported number.

B Paired Data Engine

The engine turns a corpus of seed designs into labeled sibling groups at the one-time cost C_0. Algorithm 1 states the procedure; the subsections below specify each step.

B.1 Seed curation and leak protection

Seeds are filtered to synthesizable single-file modules between 8 and 400 non-blank logic lines. The synthesizability filter is static and precedes any tool invocation: a seed is rejected if it lacks a `module`/`endmodule` pair, contains no `assign` or `always` block, or uses constructs that cannot be synthesized (`initial` blocks, delay control, system tasks, `force`/`release`, non-synthesizable types).

Leak protection runs before the engine spends any budget. Every design and every reference implementation of Pluto (Abdelatty et al., 2025b), RTLLM v2 (Liu et al., 2024) and RTL-OPT (Lu et al., 2026) is indexed, and a seed is rejected if it is a near-duplicate of any indexed design. The index is a MinHash-LSH structure over 5-token shingles of the comment-stripped, α-renamed token stream: identifiers are consistently renamed to `id0, id1, ...` before shingling, so a benchmark design that was renamed or reformatted is still caught. Parameters are 128 permutations in 16 bands and a Jaccard threshold of 0.8; exact matches are caught first by a normalized content hash. The same structure at threshold 0.95 removes intra-corpus duplicates.

Table 5: Edit generators. Counts are per seed, before deduplication and before the sibling-group cap $K = 12$.

Generator	Mechanism	Per seed
`llm_taxonomy`	LLM, one sampled pattern each	6
`rule_rewriter`	source-to-source identities	2
`corruption`	injected bugs	1

Table 6: The taxonomy of hardware optimization patterns used to condition the LLM generator. "Targets" lists the metrics a pattern primarily moves (A = area, D = delay, P = power). The risk prior is a prior probability of breaking equivalence, used only to bias sampling; the actual correctness label $z(e)$ always comes from the verification cascade.

Pattern id	Transformation	Targets	Risk prior
`arith.strength_reduction`	arithmetic strength reduction	A/D/P	0.10
`arith.tree_balancing`	operator tree balancing	D	0.12
`arith.carry_select`	adder architecture change	D	0.30
`logic.resource_sharing`	resource sharing / common sub-expressions	A/P	0.22
`logic.boolean_simplification`	boolean simplification	A/D	0.10
`logic.mux_restructuring`	priority chain → parallel case	D/A	0.25
`logic.comparator_optimization`	comparator optimization	A/D	0.15
`logic.dead_logic_removal`	dead and redundant logic removal	A/P	0.15
`logic.constant_propagation`	constant propagation and folding	A/D/P	0.12
`width.bit_narrowing`	bit-width narrowing	A/P	0.35
`seq.clock_gating`	enable-based clock gating	P	0.20
`seq.operand_isolation`	operand isolation	P	0.30
`seq.fsm_encoding`	FSM re-encoding	A/D	0.25
`seq.retiming`	register balancing / retiming	D	0.45
`seq.reset_optimization`	reset optimization	A/P	0.35
`mem.array_inference`	register file → inferred RAM	A/P	0.35
`loop.unroll_or_roll`	loop unrolling / rolling	A/D	0.30
`encoding.gray_counter`	counter / encoding transformation	D/P	0.30
`datapath.precompute`	precomputation / lookup	D/P	0.30
`datapath.pipeline_balance`	combinational depth reduction	D	0.25

B.2 Edit generators

Three generators propose edits on each seed, with the per-seed counts of Table 5.

LLM generator. Each call conditions on one *sampled pattern* from a taxonomy of 20 hardware optimization patterns (Table 6) rather than asking for "some optimization". Sampling a pattern per proposal is what makes a sibling group cover a spread of transformation families, which is the spread the listwise ranking term needs: a group of six near-identical rewrites carries almost no ordering information. Each pattern declares the metrics it primarily targets and an applicability predicate over the source, so proposals are biased toward transformations the design can actually support. The prompt states the pattern, the metric to prioritize, and four hard requirements: preserve the module name and port list byte-identically, preserve cycle-accurate behaviour, stay synthesizable, and emit the complete module rather than a patch.

Rule-based rewriters. Fifteen source-to-source transformations (Table 7) whose semantics are identical by algebraic identity. They give the engine a stream of siblings that are correct by construction but whose *PPA* effect is still unknown and must still be measured — precisely the kind of pair the differential model learns from, and a class of edits an LLM proposes only occasionally. Thirteen of the fifteen hold for arbitrary operand widths and signedness and are therefore labeled $z = 0$ without a formal proof; two (common sub-expression hoisting, and `if`/`else` to conditional expression) depend on context-sensitive width rules and are verified like any LLM proposal. All matching is performed on a comment-masked copy of the source in which comment characters are

Table 7: Rule-based rewriters. "Guar." = yes marks identities that hold for any operand width and signedness; the two "no" rewriters rely on context-dependent width rules and are verified like LLM proposals.

Rewriter	Targets	Guar.
multiply by $2^k \rightarrow$ shift	A/D/P	yes
unsigned divide by $2^k \rightarrow$ shift	A/D/P	yes
unsigned modulo by $2^k \rightarrow$ mask	A/D/P	yes
multiply by small constant $\rightarrow$ shift/add	A/D	yes
merge consecutive constant shifts	A/D	yes
De Morgan	A/D	yes
remove double negation	A	yes
remove algebraic identities	A/D/P	yes
collapse trivial conditional	A/D	yes
remove single-element concatenation	A	yes
rebalance associative chain	D	yes
zero comparison $\rightarrow$ reduction operator	A/D	yes
conditional expression $\rightarrow$ `if/else`	A/D	yes
hoist common sub-expression	A/P	no
`if/else` $\rightarrow$ conditional expression	A/D	no

Table 8: Injected faults of the corruption generator.

Mutation id	Injected fault
`bug.arith_operator`	swap + and −
`bug.logic_operator`	swap `&` and `|`
`bug.comparison`	swap comparison operator
`bug.off_by_one`	off-by-one constant
`bug.bit_range`	wrong bit range
`bug.blocking_assign`	blocking for non-blocking
`bug.reset_polarity`	inverted reset polarity
`bug.edge_polarity`	flipped clock edge
`bug.drop_else`	missing `else` branch
`bug.swap_operands`	non-commutative operand swap
`bug.constant_width`	truncated literal width
`bug.sensitivity`	incomplete sensitivity list

replaced by spaces at identical offsets, so a pattern can never fire inside a comment and a splice can never corrupt one.

Corruption generator. Twelve mutations (Table 8) covering the failure modes that actually occur in machine- and human-written RTL. They supply the negative correctness labels that train the risk head $\hat{\rho}$. Two design choices matter. First, a mutation must produce a *bug*, not a syntax error: the rewrite has to reach simulation for its failure to be a correctness signal rather than a parse signal, so every mutation is a local, type-preserving substitution. Second, a mutation is not guaranteed to be observable — a mutated branch may be unreachable under the seed testbench — so known_correct $=$ false is treated as a prior only, and the recorded $z(e)$ is whatever the real checks report.

B.3 VERIFICATION CASCADE

Candidates are checked in increasing order of cost, so that the expensive stages only ever see candidates that survived the cheap ones.

1. **Parse and interface (microseconds).** The rewrite must contain a parsable module, and it must preserve the interface of the seed: identical top module name and identical multiset of (port name, direction, width). Both ANSI and non-ANSI port styles are parsed, with direction and width inheritance for bare names, so a rewrite may legitimately switch styles.

Table 9: The synthesis flow F. One "synthesis invocation" is one Yosys mapping followed by one OpenSTA analysis, and is counted as a single call by every ledger.

Setting	Value
Synthesis, area	Yosys, `stat -liberty`
Timing, power	OpenSTA
Technology library	Nangate45 (typical corner)
Target clock period	2.0 ns, fixed per seed
Input / output delay	10% of the period
Output load	0.01 pF
Switching activity	0.1, duty 0.5
Clock / reset detection	from port names, else edge-sensitivity
Combinational designs	virtual clock at the same period
Timeout	900 s per invocation

This is the admissibility condition of the main paper, and it is enforced before a single tool second is spent.

2. **Functional simulation (seconds).** Icarus Verilog compiles the rewrite against the seed testbench and the run is classified from its log: an explicit failure marker always loses; a success marker wins over a silent run; a silent run passes iff the exit code is clean. Marker matching guards against the common false positive of a testbench printing "0 errors".
3. **Formal equivalence checking (minutes).** On a sampled subset (sampling rate 0.15 during corpus construction), the rewrite is proven equivalent to the seed. The checker is a two-stage hybrid: a Yosys `equiv_make` → `equiv_simple` → `equiv_induct` → `equiv_status` pass runs first, since structural and inductive equivalence resolves most pairs in seconds without an SMT solver; only pairs that pass returns inconclusive escalate to SymbiYosys, which proves an equivalence miter (`miter -equiv -flatten -make_assert -ignore_gold_x`) in `prove` mode at depth 20 with an SMT back end. Both stages are exposed independently, and formal checking can be disabled (simulation still gates).

A failed check at any stage is recorded as $z(e) = 1$ together with the stage that rejected it, rather than discarded. Synthesis failures on candidates that cleared verification are likewise recorded as $z(e) = 1$: a rewrite that no tool can map is not a usable optimization.

B.4 Synthesis recipe and PPA measurement

Flow, library, constraints and target period are fixed once per seed and reused for every sibling, so that the only varying factor inside a group is the edit itself. The recipe is content-addressed: its identifier is a hash of the library path, the target period, the clock and reset port names, the derived input and output delays, the load, the ABC script and the flattening flag — deliberately *excluding* the top module name, which is a property of the design rather than of the constraint set. Every emitted row carries its recipe identifier, so a corpus assembled from heterogeneous runs can be partitioned after the fact.

Metric definitions. *Area* is the total mapped cell area reported by Yosys against the target library. *Power* is the total power reported by OpenSTA under a uniform input switching activity; when the library carries no power tables the flow falls back to an area-proportional estimate and flags the measurement, so a library without power characterization cannot silently produce a meaningless third metric. *Delay* is computed as

$$\text{delay} \;=\; T_{\text{period}} \;-\; \text{WNS}, \tag{9}$$

where WNS is the worst slack on the maximum path across all corners. This is the tool-agnostic way of turning a constrained STA run into an absolute number, and it is stable for both sequential designs (register-to-register paths) and purely combinational ones (input-to-output paths under the virtual clock), which a design corpus mixes freely. Because the period is fixed within a group, differences in Eq. (9) across siblings are exactly differences in WNS.

Listing 1 Yosys mapping script. `LIB` and `PERIOD_PS` come from the seed's recipe and are identical for every sibling in the group.

```
read_verilog -sv -defer design.v
hierarchy -check -top TOP
flatten
proc
opt -full
fsm
opt -full
memory -nomap
opt -full
memory_map
opt -full
techmap
opt -full
dfflibmap -liberty LIB
opt -full
abc -liberty LIB -D PERIOD_PS
setundef -zero
clean -purge
opt_clean -purge
stat -liberty LIB
write_verilog -noattr netlist.v
```

Listing 1 gives the mapping script. Its only design-dependent argument is the top module; the library and the period enter through the recipe.

B.5 Noise-floor calibration

The ranking term must not be asked to reproduce an ordering the tools cannot resolve, so a per-metric noise floor ε_m is calibrated before the corpus is built and effects closer than ε_m are treated as ties.

The main paper calibrates ε_m by repeated synthesis of a sampled subset. The repetitions must vary something — a bit-identical rerun of a deterministic flow reproduces its numbers — so the released implementation realizes each repetition as a *semantics-preserving perturbation of the source*, which measures exactly the quantity the floor is meant to capture: the flow's sensitivity to equivalent rewrites. Three perturbations are used, each of which must leave $y(d)$ unchanged in theory:

1. renaming internal (non-port) wires and registers;
2. swapping two independent continuous assignments, whose textual order is not semantic;
3. wrapping a right-hand side in redundant parentheses.

For each of 64 sampled seeds we draw 3 perturbations, synthesize each under the seed's own recipe, and record $|\Delta_m|$ against the unperturbed measurement. The floor is the 95th percentile of the pooled $|\Delta_m|$ per metric:

$$\varepsilon_m \;=\; Q_{0.95}\Big(\big\{\,|\Delta_m(\text{perturbation})|\,\big\}\Big). \tag{10}$$

The calibrated values are written to `noise_floor.json` alongside the corpus and are consumed by the ranking loss and by every tie-aware comparison downstream. The perturbations are verified to preserve the module interface before being synthesized, so a malformed perturbation cannot inflate the floor.

B.6 Corpus composition and the cost of building it

Under the flow of Table 9, the engine turns 18,400 seed designs into 1.13×10^5 labeled pairs at a corpus cost of $C_0 = 1.33 \times 10^5$ synthesis invocations. Table 10 decomposes that budget. The decomposition follows from the quantities reported in the main paper together with the per-seed generator counts of Table 5: each seed pays exactly one base synthesis, each candidate that clears

Table 10: Decomposition of the corpus budget. Seeds, labeled pairs and C_0 are reported in the main paper; the remaining rows follow from them and from the per-seed generator counts. Rejections before synthesis contribute $z = 1$ rows at no synthesis cost.

Item	Count
Seed designs	18,400
Candidate edits proposed (9 per seed)	165,600
Base syntheses (one per seed)	18,400
Syntheses of candidates clearing verification	114,600
of which yield a usable Δ label	113,000
of which fail synthesis $\Rightarrow z = 1$	1,600
Corpus cost C_0	133,000
Labeled pairs ($z = 0$, with Δ)	113,000
Negative rows ($z = 1$)	52,600

the verification cascade pays exactly one more, and candidates rejected before synthesis cost nothing beyond simulation.

Two properties of this ledger are worth stating explicitly. First, C_0 is paid *once* and is amortized over every policy trained afterwards; it is not part of any method's training budget. Second, the synthesis cache is content-addressed on (source, recipe), so re-running a stage reproduces its measurements without paying for them again, and a cache hit is never billed to any ledger — which is what keeps "matched budget" comparisons honest under resumption.

Every emitted row carries the base source, the rewritten source, the unified diff, the context slice, Δ, z, the generator and pattern that produced it, the base and rewrite PPA vectors, the failing stage when applicable, and the recipe identifier. The corpus is therefore self-describing: a reward model can be trained from it without re-deriving anything, and the context slice and diff can be recomputed under different slicing settings from the two sources.

C Differential Reward Model

C.1 Query encoding

A query (d, e) is encoded as two inputs: a *context slice* of the base design and the edit as a unified diff.

Context slice. Let L be the lines of d and let $T \subseteq L$ be the lines the edit touches, obtained from a line-level diff (a pure insertion is anchored at its insertion point). We build an approximate signal-level dataflow graph over L: a line of the form `lhs` ⟨op⟩`= rhs` defines every identifier on the left and uses every identifier on the right, with index expressions on the left counted as reads; any other line only uses the identifiers it mentions, which covers conditions, sensitivity lists and port maps. Writing $\text{fanin}^k(\cdot)$ and $\text{fanout}^k(\cdot)$ for the lines reachable within k hops upstream and downstream on that graph, the slice is

$$S \;=\; H \;\cup\; T \;\cup\; \text{fanin}^k(T) \;\cup\; \text{fanout}^k(T), \tag{11}$$

with $k = 2$ and H the module header, port declarations and parameter declarations, which are always retained. If $|S|$ exceeds the line budget (120 lines), lines are *retained* in priority order — header lines first, then edited lines, then neighborhood lines by increasing distance to the nearest edited line — so the farthest neighborhood lines are dropped first and the header and the edited region always survive. Elided ranges are marked so the model can see that the slice is not contiguous.

Equation (11) serves two purposes. It decouples input length from design size, so a 2,000-line module and a 40-line module produce comparably sized queries; and it forces the model to learn local edit patterns that recur across designs rather than memorizing whole-design fingerprints, which is what makes the cross-family transfer of Table 2 of the main paper possible at all.

Unified diff. The edit is rendered as a unified diff with 3 lines of context, truncated to 160 lines with an explicit elision marker. The rendered query is

```
<|design|> ⟨slice⟩ <|edit|> ⟨diff⟩ <|predict|> delta(area,delay,power)
                                    risk
```

and is produced by one shared routine, so training, the RL reward and deployment-time reranking serialize a query identically. A mismatch here would silently degrade the reward signal, which is why the encoding is not duplicated anywhere in the implementation.

C.2 Architecture

The backbone is the body of Qwen2.5-Coder-3B (no language-modeling head) used as an encoder, adapted with LoRA. The pooled representation — the hidden state of the last non-padding token — feeds two linear heads: hidden $\to |M|$ for $\hat{\Delta}$, and hidden $\to 1$ for the risk logit. Head outputs are rescaled by a fixed affine transform whose scale and offset are the per-metric mean and standard deviation of Δ over the training split, so the heads start in the right range while the loss stays in interpretable log units. Heads are kept in fp32 under a bf16 backbone. Predictions are clamped to $|\hat{\Delta}_m| \le 2$ log units at inference, which rejects pathological outputs without affecting any realistic prediction.

C.3 Loss

Equation (4) of the main paper combines three terms. We give each in the form implemented.

Regression. A masked Huber loss over labeled edits and metrics,

$$\mathcal{L}_{\text{reg}} = \frac{1}{\sum_i v_i} \sum_i v_i \, \frac{1}{|M|} \sum_{m \in M} \ell_h\big(\hat{\Delta}_m(e_i) - \Delta_m(e_i)\big), \tag{12}$$

where $v_i = \mathbf{1}[z(e_i) = 0]$ and ℓ_h is Huber with $\delta = 0.05$ log units. The choice of δ is deliberate: 0.05 log units is approximately a 5% relative effect, the scale at which the main paper observes that "the effect of a typical source-level optimization" lives, so residuals larger than one typical effect are down-weighted as tool outliers rather than chased.

Tie-aware listwise ranking (Plackett, 1975; Xia et al., 2008). Let σ_m order a group from most to least improving on metric m by the *measured* effect, and let $s_i = -\hat{\Delta}_m(e_{\sigma_m(i)})$ be the score of the i-th ranked sibling (higher is better, matching the $\exp(-\hat{\Delta}_m)$ form of Eq. (3) of the main paper). Define the tie relation

$$i \sim_m j \iff \big|\Delta_m(e_{\sigma_m(i)}) - \Delta_m(e_{\sigma_m(j)})\big| < \varepsilon_m. \tag{13}$$

The loss at position i compares s_i only against candidates that are still in the urn *and* are not tied with position i:

$$\mathcal{U}_i = \{i\} \cup \{\, j > i \;:\; j \not\sim_m i \,\}, \tag{14}$$

$$\mathcal{L}^{(m)}_{\text{rank}} = -\frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \Big(s_i - \log \sum_{j \in \mathcal{U}_i} \exp(s_j) \Big), \tag{15}$$

where $\mathcal{I} = \{i < K : |\mathcal{U}_i| \ge 2\}$ skips positions at which every remaining sibling is tied with i, because such a position teaches nothing. The thresholds ε_m are the calibrated values of Eq. (10). Equation (15) reduces to the standard Plackett–Luce negative log-likelihood when $\varepsilon_m = 0$, and $\mathcal{L}_{\text{rank}}$ is its average over metrics and over the sibling groups in the batch. Setting `reward_model.tie_handling` to `none` recovers $\varepsilon_m = 0$ and orders every pair regardless of measurement resolution.

Risk. Binary cross entropy of $\hat{\rho}(e)$ against $z(e)$ over all rows of the batch, including the rejected ones. Because the batch sampler always reserves room for at least one $z = 1$ sibling per group, both classes are present in every batch.

Table 11: Reward-model architecture, loss and optimization.

Hyperparameter	Value
Backbone	Qwen2.5-Coder-3B
Heads	2 linear ($\|M\|$ and 1 outputs)
Pooling	last non-padding token
Max sequence length	2,048 tokens
LoRA rank / α / dropout	32 / 64 / 0.05
LoRA target modules	`q,k,v,o,gate,up,down`
Precision	bf16 backbone, fp32 heads
Gradient checkpointing	enabled
μ_r (ranking)	1.0
μ_z (risk)	0.5
Huber δ	0.05 log units
Ranking temperature	1.0
Tie handling	noise floor ε_m
Target rescaling	fixed affine (corpus statistics)
$\|\hat{\Delta}\|$ clamp	2.0 log units
Sibling groups per batch	2
Max group size in a batch	8
Epochs	2
Learning rate (backbone / heads)	1×10^{-4} / 1×10^{-3}
Weight decay	0.01
Warmup / schedule	3% / cosine
Gradient accumulation	8
Gradient clipping	1.0
Context slice budget	120 lines
Fan-in / fan-out depth	2 / 2
Diff budget / context lines	160 / 3

Batching. The unit of batching is the whole sibling group, never a shuffle of individual edits: comparisons inside a group cancel design-level idiosyncrasies by construction, and the group is also the unit that the within-group advantage normalization of Eq. (6) of the main paper and the evaluation protocol use. A batch is 2 groups of up to 8 siblings each, with labeled siblings preferred and room reserved for a rejected one.

C.4 Training configuration

Table 11 lists the reward-model hyperparameters. The regression and ranking terms are weighted $\mu_r = 1.0$ and the risk term $\mu_z = 0.5$: the ranking term carries full weight because it is the component the main paper credits with $+3.1$ AUC points, while the risk head is auxiliary — it acts as a pre-filter at deployment and never decides correctness.

C.5 Evaluation splits

Table 2 of the main paper reports pairwise ranking AUC on three splits, which are constructed as follows. Design families are inferred from the design's name and content (memory, FSM, arithmetic, DSP, communication, control, encoding, misc).

- **In-family.** 20% of families are held out entirely; from the remaining families, 10% of *base designs* are held out. This split tests unseen designs in familiar families.
- **Cross-family.** The held-out families. This split is family-disjoint from training: no design of a held-out family is ever seen.
- **External (RTLOpt).** The 120 triples released with CODMAS (Chang et al., 2026), loaded from a separate file and never used for training. Pairwise accuracy is reported for this split, as in the main paper.

Algorithm 2 DiffRTL

Require: seed corpus $\mathcal{S}$, flow F, budget B_{train}
Engine: edit, verify, and synthesize siblings of every $d \in \mathcal{S}$ under a shared recipe; emit $\mathcal{D}$ {one-time cost C_0}
Reward model: train f_θ on $\mathcal{D}$ by Eq. (4)
Warm start: fine-tune π_ϕ on improving edits of $\mathcal{D}$
for $t = 1, 2, \ldots$ **do**
  sample (d, λ); draw rollouts $e_{1:G} \sim \pi_\phi(\cdot \mid d, \lambda)$
  simulate $e_{1:G}$; rewards by Eq. (5)
  advantages by Eq. (6); GRPO step on ϕ
  if $t \bmod N = 0$ **and** B_{train} not exhausted **then**
    synthesize and verify A fresh rollouts {audit}
    record gap g by Eq. (7); update f_θ on audit labels
    if $g > \tau_g$ on consecutive audits **then**
      halve N; strengthen the KL anchor {harden}
    end if
  end if
end for
return π_ϕ, f_θ

Splits are design-disjoint by construction — a base design appears in exactly one split, so a sibling of a test design cannot appear in training. A further 3% of the training designs form a validation split used only for monitoring.

Pair construction. For every sibling group in a split, all pairs (e_i, e_j) of labeled siblings are enumerated, capped at 12 pairs per group so that large groups do not dominate, and labeled by which sibling has the larger measured J_λ. AUC is computed over the signed difference of predicted objectives with ties counted as 0.5; per-metric AUCs are computed analogously over $\hat{\Delta}_m$ differences, since a model can rank area well and delay badly. A pair whose measured objectives differ by less than a configurable margin can be excluded; all reported numbers use margin 0, i.e. every resolvable pair is scored.

C.6 The absolute baselines, precisely

The three contrast formulations of Table 2 of the main paper share the backbone, the training pairs, the optimizer and the schedule with the differential model. Only the rendered input and the regression target change, which is what makes the comparison a formulation comparison.

- **Absolute** renders the *edited design alone* and regresses $\log y_m(e(d))$. Downstream code consumes predictions through Eq. (8), so the induced objective is $-\sum_m \lambda_m \log y_m(e(d))$: it ranks designs, which is all an absolute predictor can offer, but it carries the design-level magnitude the task never uses — the formulation of learned absolute-PPA estimators such as MetRex (Abdelatty et al., 2025a).
- **Absolute+diff** uses the same network, evaluated twice — once on the base and once on the rewrite — and subtracts the two predictions at test time. Training regresses both rows against their own $\log y$, while the ranking and risk terms act on the difference, which is what the formulation actually consumes. Subtraction cancels shared bias but not the two independent prediction errors, which is why it recovers only part of the gap.
- **Differential w/o rank** is the full differential model trained with the regression term only ($\mu_r = 0$).

Because a group shares its base, the noise floor applies unchanged to the absolute formulation (Appendix A), so no formulation is advantaged by a different tie convention.

D Audited Surrogate-Reward Reinforcement Learning

D.1 WARM START

The policy is initialized by supervised fine-tuning on verified improving edits drawn from the engine's corpus: 3×10^4 examples of the form $\big(\text{prompt}(d, \lambda) \to \text{improved module}\big)$, restricted to edits that are formally verified and that improve J_λ under the training weights. Examples are ordered by realized gain and sampled so that the top half by gain is retained in full and the remainder is drawn uniformly, which keeps the best demonstrations without collapsing the diversity of transformation families. The prompt template is byte-identical to the one used for RL rollouts and for evaluation, so nothing about the input distribution shifts between stages. Loss is masked to completion tokens.

For RL the SFT adapter is *merged into the base weights* and a fresh LoRA adapter is attached on top:

$$\text{base} \xrightarrow{\text{merge SFT}} \text{warm start} \xrightarrow{\text{new LoRA}} \pi_\phi.$$

Disabling the RL adapter therefore recovers the warm start exactly, which is what the KL anchor is computed against — at no additional memory cost, since no second copy of the policy is held.

D.2 POLICY OBJECTIVE

Let a group of G rollouts be drawn for instance (d, λ) with rewards $r_1, \dots, r_G$ and within-group advantages $\hat{A}_i$ from Eq. (6) of the main paper. Writing π_ϕ for the policy, π_{old} for the sampling policy, π_{ref} for the warm start, and $\rho_{i,t} = \pi_\phi(a_{i,t} \mid s_{i,t}) / \pi_{\text{old}}(a_{i,t} \mid s_{i,t})$ for the token-level importance ratio, the per-token objective is

$$\begin{aligned} \mathcal{L}_{i,t} = & -\min\Big(\rho_{i,t}\hat{A}_i,\ \text{clip}(\rho_{i,t}, 1-\epsilon, 1+\epsilon)\hat{A}_i\Big) \\ & + \beta\, D^{\text{k3}}_{i,t}, \end{aligned} \tag{16}$$

with $\epsilon = 0.2$, β the KL coefficient, and the low-variance non-negative K3 estimator

$$D^{\text{k3}}_{i,t} = \exp(u_{i,t}) - u_{i,t} - 1, \qquad u_{i,t} = \log \frac{\pi_{\text{ref}}}{\pi_\phi}\Big|_{i,t}. \tag{17}$$

The loss is masked to completion tokens — everything after the first end-of-sequence token is excluded — and length-normalized per sequence before averaging over the batch, so a long rollout does not dominate a short one. With one inner epoch, $\pi_{\text{old}} = \pi_\phi$ and $\rho \equiv 1$, so Eq. (16) reduces to the plain group-relative policy gradient (Shao et al., 2024) with a KL anchor; the clipping and the ratio become active when `rl.grpo.inner_epochs` is raised above one.

A degenerate group — all G rollouts receiving the same reward, for instance all failing simulation — yields $\hat{A}_i = 0$ for all i and therefore no policy gradient. This is the correct behaviour and not a failure mode: such a group carries no comparative information.

D.3 HIERARCHICAL REWARD

The reward of Eq. (5) of the main paper — whose hierarchical structure is shared with synthesis-rewarded RL (Chen et al., 2025; Zhou et al., 2026) — is implemented as a cascade in which each tier can only reject, and only the last tier is priced by f_θ:

1. **Parse.** A completion with no extractable Verilog module receives r_{fail}; a malformed or uncompilable rewrite fails simulation trivially.
2. **Interface.** A rewrite that does not preserve the module interface receives r_{fail}, before any tool runs.
3. **Functional simulation.** Real simulation against Φ; failure receives r_{fail}. This tier is real in every DiffRTL configuration; the sole exception is the Table-3 ablation of the main paper in which $\hat{\rho}$ replaces the gate, run precisely to quantify what the real gate is worth.
4. **PPA.** One forward pass of f_θ gives $\hat{J}_\lambda(e) = -\sum_m \lambda_m \hat{\Delta}_m(e)$, and the reward is $\text{clip}(\kappa\, \hat{J}_\lambda(e))$ with scale $\kappa = 10$ and clip 5.

Algorithm 3 Audit at step t

Require: step t, interval N, batch A, threshold τ_g
if $t \bmod N \neq 0$ **or** B_{train} has fewer than A calls left **then**
  return
end if
$P \leftarrow$ up to A fresh rollouts carrying a predicted objective
$\mathcal{R} \leftarrow \emptyset$; $g \leftarrow \emptyset$
for $(d, e) \in P$ **do**
  verify e with the real tools (interface, simulation, formal)
  if verified **then**
    $J_\lambda(e(d)) \leftarrow$ real synthesis against the shared base
    $g \leftarrow g \cup \{\hat{J}_\lambda(e) - J_\lambda(e(d))\}$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(d, e, \Delta(e), z{=}0)\}$
  else
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(d, e, \perp, z{=}1)\}$
  end if
end for
report $g \leftarrow 100 \cdot \text{mean}(g)$ {Eq. (7), in pp}
update f_θ on $\mathcal{R}$ mixed with a corpus replay sample
if $g > \tau_g$ on this and the previous audit **then**
  $N \leftarrow \max(N_{\min}, \lfloor N/2 \rfloor)$; $\beta \leftarrow 2\beta$
end if

The scale κ maps log-ratio objectives, typically at most 0.1, onto $O(1)$ rewards before within-group normalization; the clip bounds the influence of a single pathological prediction. $r_{\text{fail}} = -1$ sits below the reward of any *improving* rewrite, so a failure is never preferable to an improvement; a rewrite that passes simulation but sharply degrades PPA can score below r_{fail}, which is intended — correct-but-much-worse should not outrank a clean failure signal within a group.

Two configuration switches make the ablations of Table 3 of the main paper exact rather than approximate. Setting `reward.use_simulation_gate` to false replaces tier 3 by a threshold on $\hat{\rho}$, which is the "no functional simulation gate" row. Setting `reward.source` to `synthesis` replaces tier 4 by a real synthesis call charged to B_{train}, which is the synthesis-rewarded baseline; every other part of the loop is byte-identical.

D.4 Audit protocol

Audits are the sole defense against over-optimization and the only consumer of B_{train}. Algorithm 3 states the procedure.

Four details matter.

What is auditable. Only rollouts that carry a predicted objective can be audited: a rollout that already failed simulation has no prediction to compare against, so including it would dilute g with zeros rather than measure prediction error. Equation (7) of the main paper writes the gap as a $1/A$ average over the fresh rollouts; for a rollout that fails verification or synthesis, J_λ is undefined, so the implementation takes the average over the measurable subset (its record still updates f_θ). We note this explicitly as an implementation refinement of the $1/A$ average.

What is billed. The base synthesis needed to turn a rewrite's absolute PPA into J_λ is not billed — it is a property of the instance, is shared with every other rollout on that instance, and is almost always a cache hit. The number of calls an audit reports is the ledger delta it actually caused, so a repeated rollout served from cache is free. With $N = 10$ over 200 steps and $A = 32$, the run performs exactly 20 audits and consumes $20 \times 32 = 640$ calls, which is B_{train} exactly.

How f_θ is repaired. Audit rows are exactly `EditRecords`: the same schema the engine emits, so the audit update is the same objective (Eq. (4)) on fresh data rather than a separate mechanism. Eight optimizer steps at 5×10^{-5} are taken on the audit rows mixed with a corpus replay sample

Algorithm 4 Deployment protocol for one instance

Require: instance (d_0, Φ, λ), policy π_ϕ, samples k
$y(d_0) \leftarrow \textsc{Synthesize}(d_0, R)$ {task property, unbilled}
$y(d_{\text{ref}}) \leftarrow \textsc{Synthesize}(d_{\text{ref}}, R)$ {if a reference exists}
$C \leftarrow k$ rewrites sampled from $\pi_\phi(\cdot \mid d_0, \lambda)$
$C \leftarrow \{c \in C : c$ parses, preserves the interface, passes $\Phi\}$
if reranking enabled **then**
  $C \leftarrow$ top-k' of C by $\hat{J}_\lambda$ under f_θ
end if
for $c \in C$ **do**
  formally verify c against d_0; keep only verified candidates
  $y(c) \leftarrow \textsc{Synthesize}(c, R)$ {billed to b_{eval}}
end for
return $\arg\max_c J_\lambda(c)$ over verified, synthesized candidates

of half their size, which repairs the model where the policy exploits it without forgetting the regions the audit did not cover. Setting `audit.update_reward_model` to false gives the "audit without reward-model update" arm: measurement and hardening, no repair.

Measurement without defense. When audits are disabled entirely, the same routine can still run as a *probe* charged to a separate ledger. This is how the "no audit" arm of Figure 3 of the main paper is measured: the instrument that quantifies the drift does not spend the method's budget, so the comparison across the three protocols remains at matched B_{train}.

D.5 Hyperparameters

Table 12 lists the RL configuration. Two derived quantities confirm internal consistency with the main paper. With 200 steps and 64 instances per step, the run draws $12{,}800$ instances from a pool of $18{,}400$, i.e. 0.70 epochs, matching the "about 0.7 epochs" of the main paper. With $G = 8$ that is $102{,}400$ rollouts, of which the reported 78% simulation pass rate leaves $\approx 8.0 \times 10^4$ rollouts reaching the PPA tier — matching the "about 8×10^4 predicted rewards" the main paper prices.

E Evaluation Protocol and Metrics

E.1 The shared protocol

Every method compared in the main paper is served by the identical protocol of Algorithm 4, so differences in results reflect training alone. The reward model takes no part in it; reranking is a separate, explicitly enabled mode that changes which candidates reach synthesis and never what is reported about them.

Sampling uses temperature 0.8 and top-p 0.95 with $k = 16$ candidates per task, and each configuration is run with three seeds; the main paper reports mean and standard deviation over those seeds. A task whose base design cannot be synthesized is excluded and reported as such rather than scored as a failure of the method.

E.2 Metric definitions

Pluto: eff@1. $\text{eff@1} = \frac{1}{|T|}\sum_{t \in T} c_t/n$ with $n = 16$, where c_t counts the candidates for task t that are functionally correct *and* at least as efficient as the expert reference. "Functionally correct" means the candidate parses, preserves the interface, passes simulation, and passes formal equivalence checking. "At least as efficient as the reference" is evaluated under the task's own objective:

$$J_\lambda\big(c \mid d_{\text{ref}}\big) = \sum_{m \in M} \lambda_m \log \frac{y_m(d_{\text{ref}})}{y_m(c)} \;\geq\; 0, \tag{18}$$

Table 12: Reinforcement-learning configuration.

Hyperparameter	Value
Warm start (SFT)	
Examples	3×10^4 improving edits
Formal verification required	yes
Epochs / learning rate	2 / 1×10^{-4}
Batch $\times$ accumulation	4×8
Max sequence length	4,096
Policy	
Model	Qwen2.5-Coder-7B
LoRA rank / α / dropout	32 / 64 / 0.05
Precision	bf16
KL reference	warm start (adapter disabled)
GRPO	
Steps	200
Instances per step	64
Rollouts per instance G	8
Micro-batch (sequences)	4
Learning rate / warmup	5×10^{-6} / 5 steps
Schedule	cosine
KL coefficient β	0.02 ($\times2$ on hardening)
KL estimator	K3
Clip ratio ϵ	0.2
Inner epochs	1
Sampling temperature / top-p	1.0 / 0.95
Max prompt / new tokens	3,072 / 2,048
Advantage normalization	within group, Eq. (6)
Length normalization	enabled
Gradient clipping	1.0
Reward	
Source	f_θ (model)
r_{fail}	-1.0
Objective scale κ / clip	10 / 5
Correctness gate	real simulation
Audits	
Interval N	10
Batch A	32
Budget B_{train}	640
Gap threshold τ_g	3 pp
Consecutive violations to harden	2
Minimum interval after halving	2
Reward-model update	8 steps at 5×10^{-5}
Corpus replay fraction	0.5
Formal check during audits	yes

i.e. the candidate's weighted log-PPA is no worse than the reference's. Both sides of Eq. (18) are measured under the same recipe. When a task ships no reference implementation, the comparison falls back to the base design and this is recorded.

RTLLM v2: PPA product. For each task, let $\Pi(\cdot) = y_{\text{area}} \cdot y_{\text{delay}} \cdot y_{\text{power}}$. With Π_t^{best} the product of the best verified rewrite and Π_t^{base} that of the task's own reference implementation,

$$\text{improvement} = 100 \left(1 - \exp\left[\frac{1}{|T|} \sum_t \log \frac{\Pi_t^{\text{best}}}{\Pi_t^{\text{base}}}\right]\right). \tag{19}$$

The mean is taken over the 49 synthesizable designs, as the main paper defines it; the one non-synthesizable design of the 50 is checked by simulation only and is excluded from the mean. A synthesizable task for which no candidate survives verification contributes ratio 1 (no change), the conservative implementation convention.

RTL-OPT: improved designs. The count of tasks for which the protocol returns a verified rewrite with $J_\lambda(\text{best} \mid \text{base}) > 0$.

Reward model: ranking AUC. As defined in Appendix C.5. The external RTLOpt split reports pairwise accuracy.

E.3 Reward-model reranking

With reranking enabled, f_θ scores the candidates that survived simulation and only the top $k' = 4$ are formally verified and synthesized. This is a pure cost intervention on the deployment side: it changes b_{eval}, not the measurement, and the returned rewrite is still the best *verified* one by measured PPA. The risk head can additionally drop candidates it considers likely broken before they consume a synthesis call; equivalence is still decided by the real checker for whatever survives. In the main paper this reduces per-task synthesis from 9.6 to 3.8 calls at a 0.5-point eff@1 cost.

E.4 Per-task synthesis cost

b_{eval} counts the synthesis calls a method *causes* on a deployed task: the candidate syntheses. The base and reference syntheses are properties of the task, identical for every row of Table 1 of the main paper, and are therefore measured but not billed. This is what makes the "synth./task" column comparable across methods and what the reranking row reduces.

F Baselines

F.1 Synthesis-rewarded RL (Synthesis@B)

The synthesis-rewarded baseline follows the ChipSeek-R1 recipe (Chen et al., 2025) and shares the base policy model, the warm start, the instance pool, the GRPO hyperparameters and the evaluation protocol with DiffRTL. The only change is that the PPA tier of the reward is priced by a real synthesis call rather than by f_θ, and audits are disabled. Each surviving rollout consumes one call from B_{train}; when the budget is exhausted the run stops, which is what makes $B \in \{640, 2000, 4000\}$ a genuine budget rather than a label. Under the pass rate the main paper reports, one policy step at 64 instances and $G = 8$ rollouts consumes about $64 \cdot G \cdot 0.78 \approx 400$ synthesis calls, so a budget of B calls sustains roughly $B/400$ policy steps — about 10 steps at $B = 4{,}000$. This is why more budget buys the baseline comparatively little: the budget caps the number of *updates*, not just the amount of feedback per update.

F.2 Frozen-model per-design search

REvolution (Min et al., 2026) and EvolVE (Hsin et al., 2026) are reproduced under our protocol as per-design evolutionary search over a frozen model, with every candidate priced by the real flow. One loop is shared and the two presets differ in operators and per-design budget (Table 13). The purpose of these rows is to make the cost asymmetry measurable under one protocol: a search

Table 13: Configuration of the reproduced per-design search baselines. Per-design budgets match the call counts reported in Table 1 of the main paper.

Setting	REvolution	EvolVE
Population	12	10
Generations (max)	24	24
Elites	4	3
Crossover fraction	0.33	0
Early stop (stale generations)	4	4
Synthesis budget per design	287	240

Table 14: The three cost ledgers.

Ledger	Charged by	Amortized over
C_0	paired data engine	every policy trained later
B_{train}	audits (DiffRTL) rewards (baseline)	one training run
b_{eval}	evaluation protocol	one deployed task

baseline reaches comparable quality on RTLLM v2 while spending hundreds of synthesis calls *per design*, whereas a trained policy spends its calls once, at training time, and amortizes them over every design served afterwards.

Candidates are evaluated by the same cascade as everywhere else — interface, simulation, then synthesis — so a search baseline cannot buy quality with unverified rewrites. Mutation prompts ask for one additional optimization on an elite; crossover prompts ask for an implementation combining two elites.

F.3 Frozen base and SFT only

The frozen base row evaluates the untrained policy model under the identical protocol; the SFT-only row evaluates the warm start without any RL. Both are produced by pointing the evaluator at a different checkpoint, with no other change.

G Cost Accounting and Budget Enforcement

G.1 Three ledgers

Synthesis calls are counted in three separate ledgers, and the separation is enforced in code rather than maintained by convention: there is exactly one place where a synthesis call can be charged, and the active ledger is selected by the stage that issues the call.

Table 14 names the three and states what each is amortized over.

Three rules keep the accounting honest. A cache hit is not billed, because it causes no tool work; base and reference syntheses at evaluation time are not billed, because they are properties of the task rather than of the method; and a limit is a hard cap, not a report — an audit stops early rather than overspending, and the synthesis-rewarded baseline stops training when its budget runs out. Every run writes its ledger into its result file, including the number of cache hits, so a reported budget can be audited after the fact.

G.2 Deriving the quoted cost figures

Table 15 reproduces the arithmetic behind the cost claims of the main paper from two measured rates — one synthesis call takes 1.2 min on one worker, one predicted reward takes 0.31 s on one A100 — and the run sizes of Appendix D.

Table 15: Derivation of the cost figures quoted in the main paper, from the two measured rates (1.2 min per synthesis call, 0.31 s per predicted reward) and the run sizes of Table 12.

Quantity	Value
Rollouts per run ($200 \times 64 \times 8$)	102,400
Reaching the PPA tier (at 78% pass)	$\approx 8.0 \times 10^4$
Predicted rewards, one A100	6.9 h
Predicted rewards, eight A100s	0.9 h
Same rewards priced by synthesis, serial	1,600 h
across 32 workers	50 h = 2.1 days
Audits (640 calls), serial	12.8 h
across 32 workers	0.4 h
Ratio	**125×**
Corpus C_0 (1.33×10^5 calls), serial	2,660 h
across 32 workers	83 h = 3.5 days

The asymmetry the table exposes is the point of the method. A synthesis call spent as a reward, or spent on a single deployed design, is consumed once; the same call spent auditing the differential model keeps pricing every later rollout. That is how 640 calls end up beating budgets six times larger, and why DiffRTL at 500 synthesis calls already exceeds the synthesis-rewarded baseline at 8,000 — at least 16× fewer calls at equal quality.

G.3 What the reward model can and cannot do

The division of labor is strict: f_θ decides only which rollouts are reinforced, while the equivalence and the PPA of anything reported are measured by the real tools. An inaccurate reward model can therefore waste training compute at worst; it cannot fabricate a result. This property is a consequence of where f_θ is allowed to appear, and the released implementation enforces it operationally: analytic stand-ins for the tools exist so that the pipeline can be exercised without an EDA installation, they announce themselves in the logs, they stamp a flag into every result file, and any command invoked with `--require-real` refuses to run with them.

H Additional Analysis of the Audit Trace

H.1 What the gap measures

Equation (7) of the main paper subtracts a measured quantity from a predicted one on the *same* rollouts, so the audit gap is a direct estimate of the reward model's bias in the region the policy currently occupies — not its average error over a static test set. This distinction is what makes the gap a usable control signal: a model with excellent held-out AUC can still have a large gap if the policy has drifted into a corner of the edit space the corpus covered poorly, and it is exactly that corner the audit then repairs.

Because the audit measures rollouts drawn from the current policy, the trace is also a measurement of over-optimization (Gao et al., 2023) against an executable oracle rather than against a proxy. The three protocols of Figure 3 of the main paper isolate the three things the routine of Appendix D.4 does:

- **No audit** (measurement only, charged to a separate ledger). Predicted improvement climbs to 8.7% while realized quality peaks near step 68 and then decays — a 5.7 pp account of over-optimization. The policy is optimizing the predictor, not the design.
- **Audit without reward-model update** (measure and harden, no repair). The drift slows and one hardening event occurs near step 136, visible as a kink in the trace, yet the gap still reaches 4.0 pp. An alarm slows what only repair can stop.
- **Full audit** (measure, repair, harden). The gap stays below 1 pp and hardening is never triggered. A single audit removes about two thirds of the accumulated drift (0.45 → 0.15 pp).

Table 16: Reward-model ranking quality and final policy quality scale together. Both columns are reported in the main paper; subsampling is by whole sibling group, so every subset remains a set of controlled experiments.

Training pairs	In-family AUC	Pluto eff@1
1.0×10^4	0.741	48.1
3.0×10^4	0.789	51.2
1.13×10^5	0.824	53.6

The three protocols end at 47.2, 49.0 and 53.6 eff@1, so the ordering of final quality matches the ordering of measured drift.

H.2 The hardening state machine

Hardening is deliberately hysteretic: it fires only after τ_g is exceeded on two *consecutive* audits, and a single clean audit resets the counter. A single noisy audit — 32 rollouts is a small sample — should not permanently halve the interval and double the KL coefficient. When it does fire, both responses push in the same direction: halving N buys more measurements per unit of training, and doubling β holds the policy closer to the warm start while f_θ is being repaired. Halving is floored at $N_{\min} = 2$ so the schedule cannot degenerate into auditing every step and exhausting B_{train} early.

H.3 Reading a run

Three logged quantities are sufficient to diagnose a training run, and all three are written per step:

- `pass_rate` — the fraction of rollouts clearing the real simulation gate (78% in the main paper's runs). A collapse indicates a warm start that is too weak or a KL anchor that is too loose, and it is also what the "no simulation gate" ablation destroys: candidate pass rate falls from 52% to 31% and eff@1 to 33.5, below the frozen base. A predicted gate invites the broken code that a real gate rejects.
- `audit/gap_pp` — Eq. (7). Rising means the policy is exploiting f_θ.
- `budget/train_calls` — must never exceed B_{train}; it is enforced, not merely reported.

H.4 Reward-model quality and policy quality

The data-scaling rows of Table 3 of the main paper pair a reward-model measurement with a policy measurement on the same corpus subsets, which ties the two together directly (Table 16). Ranking quality and policy quality move together, and neither has saturated at 1.13×10^5 pairs — the returns are diminishing but not exhausted, which is the expected shape if the corpus is still the binding constraint rather than the model capacity.

Subsampling for these rows keeps *whole sibling groups* rather than individual edits. Sampling edits would break the controlled-experiment property a group provides and would make the smaller corpora incomparable to the full one.

I Implementation and Reproducibility

I.1 Software and hardware

All experiments use one open flow: Yosys for synthesis and area, OpenSTA for timing and power, Icarus Verilog for functional simulation, and Yosys `equiv_*` with SymbiYosys escalation for formal equivalence checking, under the Nangate45 library. Training runs on eight NVIDIA A100 GPUs with 32 parallel synthesis workers.

Table 17: Command-line interface. `pipeline` runs stages 1–5 in order.

Stage	Command	Output
1	`engine`	labeled pairs, C_0
2	`rm-train`	f_θ
3	`sft`	warm start
4	`rl`	π_ϕ, audit trace
5	`evaluate`	benchmark scores
—	`rm-eval`	ranking AUC
—	`baseline`	Synthesis@B, search
—	`report`	collated tables
—	`doctor`	environment preflight

I.2 Artifact structure

The implementation is released as a single Python package with a command-line interface whose subcommands correspond one-to-one to the stages of Algorithm 2 (Table 17). Every stage is independently runnable, resumable, and configured by one composable YAML tree in which unknown keys are a hard error, so a typo in a sweep fails immediately rather than silently doing nothing.

I.3 Determinism

A single seed controls Python, NumPy and PyTorch random state; the evaluation protocol is repeated with three seeds and the main paper reports mean and standard deviation over them. Sampling from a language model, thread scheduling in the tool pool, and GPU non-determinism make runs statistically rather than bitwise reproducible — which is why every reported number is a mean with a standard deviation. The synthesis cache is content-addressed, so re-running a completed stage reproduces its measurements exactly.

I.4 Extension points

Every axis a follow-up study is likely to change is a registry entry selected by name from the configuration, so substituting one requires no change to calling code: synthesis flows, simulators, equivalence checkers, seed providers, edit generators, reward-model formulations, benchmarks, generation backends and search baselines. In particular, a commercial synthesis flow or a different technology library enters by implementing a single method, and the corpus interface is one line of JSON per labeled pair, so a reward model can be trained on data produced by an entirely different engine.

I.5 Sanity infrastructure

Beyond the reported experiments, the artifact ships a test suite that pins the properties the method's argument depends on: that the objective identity of Eq. (8) holds, that the listwise loss is minimized by the correct ordering and is invariant to flips below the noise floor, that a degenerate rollout group produces no policy gradient, that the KL estimator is non-negative, that the completion mask excludes everything after the first end-of-sequence token, that rule-based rewrites preserve simulated behaviour, that injected faults are caught, that budgets cannot be overspent, and that the metric definitions of Appendix E compute what they claim. A tool-free configuration exercises the entire pipeline — engine, dataset, reward interface, the full RL loop with audits and budget accounting, and the deployment protocol — in under a minute on a machine with no GPU and no EDA installation, which makes the plumbing verifiable independently of the measurements.