Instruction-Conditioned Motion Control (ICMC) for Robotic Arms: VLA Decomposition and Reinforcement Learning Tuning

Anonymous ACL submission

Abstract

Generating reliable robot control programs from natural language remains brittle: one-shot prompting yields code that fails at execution from missing terminations, API misuse, and grounding errors. We introduce Instruction-Conditioned Motion Control (ICMC), a framework that decomposes robot code generation into structured motion-control parameters, an interpretable instruction stage, and a separate code-synthesis stage verified in physics simulation. Paired with reasoning multimodal models, ICMC reaches 100% pick-and-place success with o1 and 60% with o4-mini. To isolate what drives these gains, we ablate the reasoning component, replacing both stages with open-weight, text-only Qwen models trained via Group Relative Policy Optimization (GRPO). Naive GRPO collapses 32.0% to 11.7% per-sample success in a single update; we trace this to reward variance from the stochastic downstream code generator, and contribute a scheme (multi-sample denoising, conservative learning rate, multi-epoch GRPO) that lifts per-sample success from 28.1% to 39.4% over 20 steps. Sub-ablations decompose each component. ICMC's reliability tracks the underlying model's reasoning capability, and the scheme extends it to open-weight, text-only settings.

1 Introduction

Large language models (LLMs) increasingly serve as high-level controllers for robotic manipulation, mapping natural-language task specifications to executable control logic. Vision–Language–Action (VLA) architecture (Brohan et al., 2022; Zitkovich et al., 2023; Driess et al., 2023; Kim et al., 2024) integrates visual perception, natural language processing, and motor control into unified pipelines. The program-synthesis approaches (Liang et al., 2023; Singh et al., 2022) have LLMs compose predefined primitives into manipulation programs. Yet generation becomes brittle (Vemprala et al., 2024; Liang et al., 2023; Singh et al., 2022) since one-shot prompting often produces syntactically valid code that fails at execution due to missing termination conditions, incorrect API usage (Liang et al., 2023; Singh et al., 2022), or grounding errors that surface only in simulation (Vemprala et al., 2024; Wang et al., 2023), motivating downstream methods that ground motion in 3D affordances (Huang et al., 2023) or structured spatial primitives (Liu et al., 2022b); rearrangement benchmarks report similar specification-to-execution gaps (Mees et al., 2022; Batra et al., 2020; Bharadhwaj et al., 2024; Zeng et al., 2021; Gkanatsios et al., 2023).

Prior work approaches this gap from several angles: end-to-end VLA (Brohan et al., 2022; Zitkovich et al., 2023; Driess et al., 2023; Kim et al., 2024) that maps observations to actions but needs demonstration data; iterative-refinement and affordance-grounded methods (Huang et al., 2022; Ahn et al., 2022; Song et al., 2023); structured prompting (Liang et al., 2023; Singh et al., 2022; Vemprala et al., 2024; Huang et al., 2023); and RL from execution feedback (Ma et al., 2024; Kwon et al., 2023; Shao et al., 2024; Ouyang et al., 2022). Two gaps remain. First, existing pipelines fold instruction understanding and code synthesis into a single step, so execution failures entangle planning, synthesis, and grounding errors: the same upstream mistake recurs across full-program retries. Second, in two-model LM pipelines, an upstream policy trained against a stochastic downstream sampler receives rewards that entangle upstream quality with sampling probability; standard methods (e.g., GRPO (Shao et al., 2024)) treat this reward noise as a learning signal, collapsing the upstream policy in a single update and rendering instruction-level RL infeasible without explicit variance control.

To fill the gap, we propose the Instruction-Conditioned Motion Control (ICMC) framework to address the gaps. In this study, we investigated a few RQs. RQ1: Does decomposing code gen-

eration into a structured instruction layer plus a separate code-synthesis stage improve reliability when paired with reasoning multimodal models? RQ2: Are ICMC's RQ1 gains driven by the structural decomposition or by the reasoning capability of the underlying models? RQ3: Which components of the resulting RL scheme drive stability and lift?

We make three contributions. First, we introduce *Instruction-Conditioned Motion Control* (ICMC), a three-stage framework that reaches 100% pick-and-place with OpenAI's o1 model (OpenAI, 2024) when paired with reasoning multimodal models. Second, we identify a reward-variance bottleneck in two-model LM pipelines, in which single-sample variance from the stochastic downstream sampler dominates between-instruction variance and collapses GRPO from 32.0% to 11.7% in one update; we contribute a scheme (multi-sample denoising, conservative learning rate, multi-epoch updates) that lifts per-sample success from 28.1% to 39.4% over 20 steps. Third, sub-ablations decompose each scheme component, showing that ICMC's reliability comes from the underlying reasoning capability rather than the decomposition itself, a finding that generalizes to any two-model LM pipeline.

2 Related Work

2.1 Vision–Language–Action Robotics

VLA systems map perception and language directly to robot actions: CLIPort (Shridhar et al., 2022), RT-1/RT-2 (Brohan et al., 2022; Zitkovich et al., 2023), PaLM-E (Driess et al., 2023), and OpenVLA (Kim et al., 2024) train end-to-end transformers on large robot-demonstration corpora, with generalist agents like Gato (Reed et al., 2022) adopting similar token-prediction backbones across robotic and non-robotic tasks, while affordance-grounded methods (SayCan (Ahn et al., 2022), Inner Monologue (Huang et al., 2022)) score language outputs against learned feasibility models or use language feedback for closed-loop reasoning. These approaches typically require substantial demonstration data or tightly coupled architectures.

2.2 Language Models for Robot Code Generation

An alternative paradigm uses LLMs to generate robot control code rather than actions directly: Code-as-Policies (Liang et al., 2023) composes primitives into executable programs, ProgPrompt (Singh et al., 2022) adds structured templates to reduce hallucinated API calls, Chain-of-Code (Li et al., 2024) interleaves natural-language reasoning with executable Python (used as an experimental baseline here; Table 2), ChatGPT for Robotics (Vemprala et al., 2024) explores iterative refinement on execution feedback (also used as an experimental baseline), VoxPoser (Huang et al., 2023) generates 3D value maps for motion planning, LLM-Planner (Song et al., 2023) studies few-shot grounded planning, and Voyager (Wang et al., 2023) pairs code generation with a skill library. These methods rely on prompt engineering and static verification, leaving generated programs prone to subtle logical errors and brittle termination conditions.

2.3 Reinforcement Learning for Language Model Policies

Reinforcement learning from human feedback (RLHF) (Ouyang et al., 2022) with proximal policy optimization (PPO) (Schulman et al., 2017) optimizes language models against human preferences. The low-rank adaptation (LoRA) (Hu et al., 2022) makes the fine-tuning more parameter-efficient. The group relative policy optimization (GRPO) (Shao et al., 2024) introduces group-based advantage normalization that we adopt below; recent fine-tuning SDKs such as Tinker (Machines, 2026) package these primitives behind a managed API. In robotics, Eureka (Ma et al., 2024) and related reward-design work (Kwon et al., 2023) use LLMs to specify reward functions, but the policy itself remains conventional.

3 Method

3.1 Problem Formulation

We consider instruction-conditioned motion control for robotic manipulation: given a task description τ and visual observation o of the workspace, generate a motion control program c that completes the task when executed in a physics-based simulator. Rather than generating code directly, we introduce an intermediate representation (motion control instructions encoding task structure, control parameters, and execution logic) and treat instruction generation as a policy optimization problem,

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{(\tau,o)\sim\mathcal{D}}[R(\pi(\tau, o), \tau)], \quad (1)$$

where $R(c, \tau) \in \{0, 1\}$ is the binary success reward verifying that c completes τ under a position-

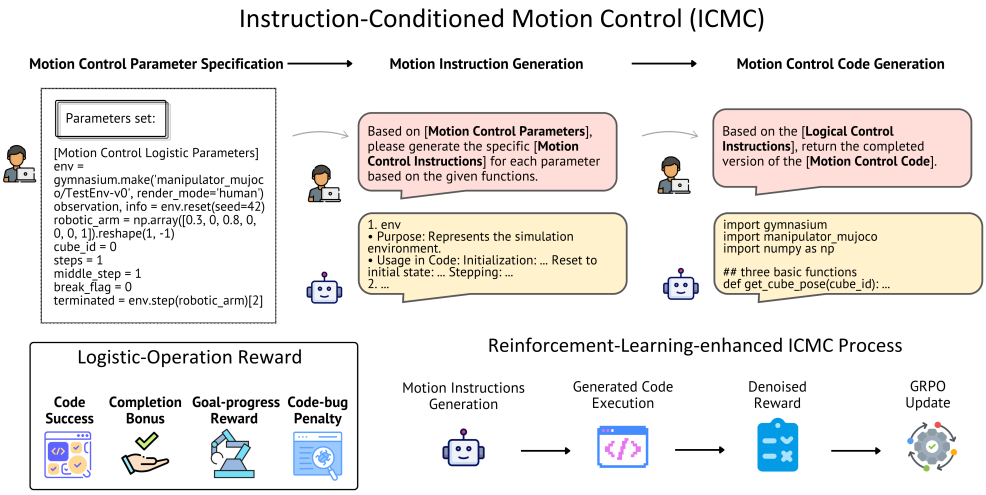

Figure 1: **ICMC with RL.** *Top:* three-stage pipeline (parameter specification, motion-instruction generation, code generation) verified in MuJoCo by position tolerance. *Bottom left:* dense reward combines binary success, static-bug penalty, goal-progress, and completion bonus. *Bottom right:* RL on the instruction policy (Section 6) samples $M{=}5$ codes per instruction, averages into a denoised reward, and applies a GRPO update; our scheme ($M{=}5$, conservative LR $1{\times}10^{-6}$, multi-epoch $E{=}3$) is ablated in Section 7.

tolerance metric (all objects within ϵ m of their targets), and $\mathcal{D}$ is a distribution over tasks. We use $\epsilon = 0.1$ for RQ1 prompting evaluations and $\epsilon = 0.05$ for the RQ2/RQ3 RL experiments, matching the reward components in Equation (10).

3.2 Instruction-Conditioned Motion Control (ICMC)

As illustrated in Figure 1, ICMC decomposes motion control generation into three stages.

Stage 1: Motion Control Parameter Specification We define motion-control parameters (simulation state, robot configuration, object identifiers, and execution variables such as step counters and termination flags) that ground instruction generation in the concrete execution context.

Stage 2: Motion Instruction Generation A language model π_{instr} generates structured motion instructions

$$l = \pi_{\text{instr}}(\tau, o), \tag{2}$$

covering perception, manipulation (grasp/move/place), and termination conditions. The instructions serve as an interpretable abstraction between task understanding and code synthesis.

Stage 3: Motion Control Code Generation and Verification A code model π_{code} synthesizes executable code

$$c = \pi_{\text{code}}(\tau, o, l) \tag{3}$$

over a fixed robot API (primitives in Appendix A.2). Task success is verified:

$$R(c, \tau) = \mathbf{1}\left[\max_i \|p_i^{\text{final}} - p_i^{\text{target}}\|_2 < \epsilon\right], \tag{4}$$

where p_i^{final} and p_i^{target} are the final and target positions of object i. Together, π_{instr} and π_{code} implement the policy π of Equation (1) via composition, $\pi(\tau, o) = \pi_{\text{code}}(\tau, o, \pi_{\text{instr}}(\tau, o))$; this decomposition supports failure-mode analysis and provides a natural interface for learning from execution feedback.

3.3 Reinforcement Learning over Motion Instructions

When prompting alone is insufficient, we apply RL to the instruction policy π_θ while keeping the code generator π_{code} frozen. We treat instruction generation as a contextual decision-making problem with state $s = (\tau, o)$, action $a = l$, and reward $r = R(\pi_{\text{code}}(\tau, o, l), \tau)$ from physics-based execution.

We use GRPO-style policy gradients with group-based advantage normalization (Shao et al., 2024). For each task, we sample K instruction rollouts $\{l_1, \ldots, l_K\}$, generate code for each, and compute rewards through simulation. The advantage for rollout k is

$$A_k = r_k - \bar{r}, \quad \bar{r} = \frac{1}{K}\sum_{j=1}^{K} r_j, \tag{5}$$

and the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{k:A_k\neq 0} A_k \nabla_\theta \log \pi_\theta(l_k|s)\right], \quad (6)$$

where the $A_k \neq 0$ filter focuses updates on rollouts that differ from the group mean. Group-based normalization yields both positive and negative gradients within each batch, enabling learning from sparse binary rewards (Appendix A.7 expands the per-token score function and the gradient-sign behavior).

4 Experimental Setup

Environment and Pipeline MuJoCo (Todorov et al., 2012) with a UR5e 6-DOF arm, a parallel gripper, and three red cubes, exposed via the Gymnasium API (Towers et al., 2026); the robot API primitives are listed in Appendix A.2. The three-stage ICMC pipeline of Section 3.2 is realized with a state machine over the variables `steps`, `middle_step`, `cube_id`, and `break_flag`, and a prompt template detailed in Appendix A.

Tasks Three categories of increasing complexity. *(1) Pick-and-Place:* pick all 3 cubes and place them in a designated disposal area. *(2) Shaping:* arrange 8–13 cubes into a target letter, evaluated by coordinate accuracy $\text{Acc}_{\text{coord}}$ (whether LLM-generated coordinates match ground truth within ϵ_{coord}). *(3) Reshaping:* transform one letter pattern into another, evaluated under both *plan-given* (Hungarian-optimal plan supplied) and *plan-generation* (LLM produces `remove_idx`, `add_idx`, and `mapping` itself) settings.

Benchmark We formulate shaping and reshaping as goal-conditioned tabletop rearrangement (Batra et al., 2020; Zeng et al., 2021; Shridhar et al., 2022). The benchmark uses 12 character targets covering five geometric motif families (linear strokes, junctions, diagonals, enclosure, bilateral symmetry; full list in Appendix C). Targets sit on a 7×7 anchor grid (step 0.031 m) within a 0.2×0.2 m workspace. Reshaping is evaluated over all $\binom{12}{2} = 66$ unordered pairs, stratified by a composite difficulty metric

$$D(A, B) = \Delta(A, B) + 6\,\mathbf{1}_{\text{encl}} + 3\,\mathbf{1}_{\text{sym}}, \quad (7)$$

where $\Delta(A, B) = |A \triangle B|$ is the symmetric-difference edit distance, and the enclosure/symmetry penalties capture non-local topology and bilateral-symmetry changes (Liu et al., 2022b,a; Gkanatsios et al., 2023). The $(1, 6, 3)$ calibration achieves Spearman $\rho = 0.92$ against the Hungarian-algorithm operation count. Pairs bucket into Easy ($D \leq 8$), Medium ($9 \leq D \leq 14$), Hard ($15 \leq D \leq 19$), and Extreme ($D \geq 20$).

Table 1: Main results: one-shot baseline vs ICMC. o1 reshaping uses 5 runs on the M→I pair; o4-mini reshaping uses 20 trials on the M→I pair (D=20, Extreme bucket), matching Table 2. Pick-and-place: o1 = 5 runs, o4-mini = 20 runs. Wilson 95% CIs for every cell are reported in Appendix F.

Task	Model	One-shot	ICMC
Pick-and-Place	o1	100%	**100%**
Pick-and-Place	o4-mini	35%	**60%**
Shaping ($\text{Acc}_{\text{coord}}$)	o1	**100%**	
Shaping ($\text{Acc}_{\text{coord}}$)	o4-mini	**100%**	
Reshaping	o1	20%	**60%**
Reshaping	o4-mini	25%	**15%**
Reshaping (plan-gen)	o1	80%	
Reshaping (plan-gen)	o4-mini	**100%**	

Models *RQ1:* reasoning models o1 (`o1-2024-12-17`) and o4-mini drive both ICMC stages; GPT-4o via LangChain (temperature 0.7) parses pipeline outputs. *RQ2–RQ3:* a trainable Qwen3.5-35B-A3B (35B-parameter mixture-of-experts model with 3B active per token) instruction policy with LoRA (rank 32, α=64) is paired with a frozen Qwen3-Coder-Next code generator served via OpenRouter; GRPO is run with TRL (von Werra et al., 2020).

Evaluation We report three metrics: *success rate* (all cubes within ϵ of targets), *position error* (mean L2 distance), and *execution time* (valid runs 10–120 s; outside this range counts as failure). We use $\epsilon = 0.1$ m for RQ1 prompting evaluations and $\epsilon = 0.05$ m for the RQ2/RQ3 RL experiments, matching the reward tolerance in Equation (10).

5 RQ1: Effectiveness of Instruction-Conditioned Motion Control

We evaluate whether ICMC enables reliable motion control without learning by comparing one-shot code generation against the structured ICMC pipeline across multiple model architectures (5 independent runs; Table 1).

Headline Results ICMC with o1 reaches 100% pick-and-place success (per-cube position errors in Appendix B.3) and 60% on plan-given reshaping. The smaller o4-mini benefits more from the structured decomposition: pick-and-place rises from 35% (one-shot) to 60% (ICMC), a 1.7× lift; o1 reshaping rises from 20% to 60% (3×). The dominant failure mode in one-shot baselines is *early exit*, where generated code terminates before completing all manipulation steps. ICMC's intermediate instruction stage is therefore most valuable for smaller models and for complex multi-phase tasks.

Plan-Generation Reshaping Both reasoning models can replace the Hungarian Algorithm for transformation planning on the M→I pair: o1 produces valid plans in 4/5 runs (80%) and o4-mini in 5/5 (100%).[1] o4-mini's plans differ from the Hungarian optimum but achieve *shorter* total movement distance (0.199 vs. 0.314 units), suggesting smaller models can perform spatial reasoning for well-specified planning tasks.

Failure Analysis and the 50% Ceiling The gap between 75% code validity (the generated Python runs without a Python-level crash within the 120-s timeout) and 15% reshaping success in o4-mini stems from runtime failures: the model frequently modifies the predefined state-machine sub-task functions (`formulate_a_pattern`, `dispose_dummy_cubes`, `execute_remap`) in ways that break Python scoping or the API contract. Two targeted fixes (amended sub-task functions and a constrained code-generation prompt) raise success to 50% (Table 2), but both are task-specific and do not generalize to new manipulation scenarios.

Comparison with Outside Baselines Table 2 additionally evaluates two outside baselines on the same M→I reshaping pair, same o4-mini backbone, and same robot API as the ICMC rows: Chain-of-Code (Li et al., 2024) (single-shot, interleaved natural-language reasoning and executable Python) and a ChatGPT-for-Robotics-style iterative self-correction loop (Vemprala et al., 2024) (N=3 retries on structured execution feedback consisting of a success flag, per-cube L2 distances, and the Python traceback). Chain-of-Code reaches 20% task success with 55% code validity, above unmodified Baseline ICMC (15%/75%) on this pair but still well below ICMC with both targeted fixes (50%). Iterative self-correction collapses to 5% task success with only 15% final-attempt code validity: after three rounds of feedback on this Extreme-bucket pair, o4-mini exhausts its visible-output budget on reasoning tokens and 16 of 20 final attempts return no parseable code at all. Neither isolated prompting baseline reproduces the lift produced by ICMC's structural decomposition combined with its task-specific fixes, and the iterative-feedback strategy in particular degrades sharply on hard reshaping, where the policy-correction signal lags behind the spatial complexity of the target.

Table 2: Reshaping ablation and outside baselines for o4-mini on the M→I pair (D=20, Extreme bucket; n=20). **ICMC rows**: Baseline ICMC and three targeted-fix ablations. **Outside-baseline rows**: Chain-of-Code (Li et al., 2024) (single-shot) and ChatGPT-for-Robotics-style iterative self-correction (Vemprala et al., 2024) (N=3 retries; iterative row reports *final*-attempt status). Wilson 95% CIs in Appendix F.

Configuration	Code Valid	Task Success
Chain-of-Code	55%	20%
Iterative correction	15%	5%
Baseline ICMC	75%	15%
+ Amended functions	100%	20%
+ Constrained prompt	70%	35%
+ Both	90%	**50%**

Bridge to RQ2/RQ3 The 50% ceiling is reached only with task-specific amendments, leaving open whether ICMC's gains come from the structured decomposition itself or from the underlying reasoning multimodal capability. RQ2 (Section 6) ablates the reasoning component by replacing both stages with open-weight, text-only models trained via RL; RQ3 (Section 7) decomposes the resulting scheme.

6 RQ2: Ablating the Reasoning Multimodal Component

RQ1 showed that ICMC, paired with reasoning multimodal models, reaches high task success. *What drives these gains: the structured decomposition itself, or the capabilities of the underlying reasoning model?* To isolate this, we replace the code generator with an open-weight text-only

[1] All reshaping sub-experiments in this paper (one-shot baseline, ICMC execution, the ICMC ablations in Table 2, the outside baselines in Table 2, and plan-generation in Tables 9 and 10) use the M→I direction on the M↔I pair. The composite difficulty metric is direction-symmetric ($D(\mathrm{M},\mathrm{I}) = D(\mathrm{I},\mathrm{M}) = 20$), so the M→I evaluation characterizes the unordered M↔I pair, which sits in the Extreme bucket of Table 13.

model (Qwen3-Coder-Next) and the instruction policy with a trainable text-only model (Qwen3.5-35B-A3B), keeping the rest of the ICMC pipeline fixed. The resulting *two-model pipeline*, a trainable instruction policy composed with a frozen stochastic code generator, retains the decomposition but discards multimodal reasoning. We ask whether RL on the upstream policy can recover the lost performance and report the best scheme we found; RQ3 (Section 7) explains why each component matters.

Two-Model Reward Variance The instruction policy π_θ produces an instruction l, and the frozen code generator samples $c \sim \pi_{\text{code}}(\cdot \mid \tau, l)$ stochastically. The reward $r = R(c, \tau) \in \{0, 1\}$ is therefore a noisy single-sample estimate of the true instruction quality

$$q(l) := \mathbb{E}_{c \sim \pi_{\text{code}}(\cdot \mid \tau, l)}[R(c, \tau)], \tag{8}$$

with variance $\text{Var}[r \mid l] = q(l)(1 - q(l))$. When this within-instruction variance dominates the between-instruction variance in $q(l)$, GRPO advantages $A_k = r_k - \bar{r}$ stop reflecting instruction quality, and a single update can fit downstream sampling noise. Empirically (Section 7.1), under standard hyperparameters, the policy collapses from 32.0% to 11.7% in one update.

Stabilization Scheme We address this variance source through three coupled modifications to the standard GRPO procedure. First, we replace the per-instruction reward with a multi-sample Monte Carlo estimator: for each instruction l_k, we draw M=5 independent code candidates and use the averaged reward

$$\hat{r}_k = \frac{1}{M} \sum_{m=1}^{M} R(c_k^{(m)}, \tau), \tag{9}$$

which is an unbiased estimator of $q(l_k)$ with variance reduced by a factor of M. Second, we adopt a conservative learning rate $\eta = 1{\times}10^{-6}$, forty times smaller than the naive baseline, bounding the per-step KL drift so that residual noise cannot destabilize the policy. Third, we perform multi-epoch updates (E=3 gradient passes per denoised rollout batch), amortizing the $M\times$ rollout cost across multiple optimizer steps.

We instantiate the scheme under two reward functions: **denoised binary**, $\hat{r}_k = \frac{1}{M}\sum_m \mathbb{1}[\text{success}_m]$; and **denoised dense**, the four-component reward

$$r_m = \mathbb{1}[\text{success}_m] - \lambda V_{\text{bug}}^{(m)} + \alpha r_{\text{goal}}^{(m)} + \beta r_{\text{comp}}^{(m)}, \tag{10}$$

with $\lambda = 0.3$, $\alpha = \beta = 0.25$ (component definitions in Appendix A.10). Both runs use K=32, M=5, E=3, $\eta = 1{\times}10^{-6}$, 20 GRPO steps (3,200 executions). The pipeline is text-only: both models receive textual scene descriptions and produce text outputs (no RGB), making the comparison against RQ1's multimodal pipelines a clean test of the role of perception and reasoning.

Table 3: **RQ2 main results.** Per-sample binary success rate over 20 GRPO steps with the denoised conservative scheme (M=5, η=$1{\times}10^{-6}$, E=3, K=32). Q1 / Q4 are averages over steps 1–5 and 16–20.

Reward	Step 1	Step 20	Q1	Q4	Δ rel.
Denoised binary	28.1%	**39.4%**	31.6%	35.2%	+40%
Denoised dense	29.7%	**38.0%**	30.0%	33.7%	+28%

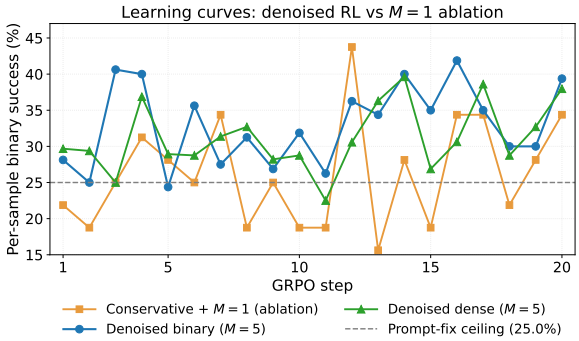

Figure 2: **Per-step learning curves.** The denoised binary (M=5) and denoised dense (M=5) runs trend upward across 20 GRPO steps, staying above the non-RL prompt-fix ceiling (dashed line). The M=1 ablation (conservative LR + E=3, denoising removed) avoids the step-2 collapse of naive GRPO (Section 7.1) but exhibits wider variance and lower Q4 (30.6% vs. 35.2%).

Results Table 3 reports per-sample binary success rates aggregated over all $K{\times}M = 160$ samples per step.

Both reward variants produce stable, upward trajectories (Figure 2): per-sample binary success rises 28.1%→39.4% under the binary reward and 29.7%→38.0% under the dense reward, with Q4 exceeding Q1 in both runs and the worst step-1→step-2 dip at −11% relative, vs. −63% under naive hyperparameters (Section 7.1). Peak per-step success reaches 41.9% (binary, step 16), exceeding the non-RL prompt-engineering ceiling of 25.0% (Section 7.2). Crucially, these peaks remain well

below the RQ1 reasoning-model pipelines (60% o4-mini, 100% o1; Table 1): the structured decomposition alone is not sufficient to close the gap left by removing the reasoning multimodal component.

7 RQ3: Sub-Ablations of the Text-Only RL Scheme

RQ2 established that, after removing reasoning multimodal models, our targeted RL scheme produces a stable but bounded recovery. RQ3 decomposes that finding further with four sub-ablations, explaining which components of the scheme matter and how they compare to non-RL baselines. Throughout, "per-sample binary success" denotes the empirical fraction of code samples that pass the position-tolerance check, computed across all $K \times M$ samples per step (or K samples for $M{=}1$).

7.1 Naive GRPO Collapses Without the Scheme

Nine baseline configurations under standard GRPO hyperparameters ($\eta = 4{\times}10^{-5}$, $E{=}1$, $M{=}1$), spanning $K \in \{32, 128\}$, 5–20 step horizons, both binary and dense rewards, and both condensed and raw instruction formats, all fail to improve. Without the prompt fix, RL stays at the no-RL baseline (5.8% mean vs. 6.6% baseline at $K{=}128$, binary, 20 steps). With the prompt fix, RL fluctuates around the 25.0% ceiling without trending (31.9% binary, 30.2% dense, both with ∼20 pp range reflecting noise rather than learning). Our scheme (Section 6) is the only configuration with a sustained upward trajectory: per-sample binary success rises 28.1%→39.4% under the binary reward and 29.7%→38.0% under the dense reward, both stabilizing above the prompt-fix ceiling (Figure 2; per-configuration outcomes in Table 7).

The diagnostic case is Exp 9 (Table 8): a single GRPO update at $\eta = 4{\times}10^{-5}$ drops per-sample success from 32.0% to 11.7% (−63% relative), consistent with the two-model variance argument in Section 6. The error-category distribution across baselines is essentially invariant to RL training, confirming that the policy does not learn to fix the dominant failure modes under naive hyperparameters.

7.2 Prompt Engineering Establishes a Strong Non-RL Ceiling

Before evaluating any RL scheme, we measured what prompt engineering alone achieves on the code-gen prompt. A one-paragraph API-documentation addendum addressing the three most common code-gen misunderstandings raises per-sample success from 6.6% (no-RL baseline) to 25.0% (prompt fix only) on the single-pattern `pick_and_place_M` task used for the RL experiments; an independent two-seed reproduction confirms the same baseline at 26.2% ± 1.7 pp.[2] We apply this addendum uniformly to all RQ2 and RQ3 runs, so the 25.0% *prompt-fix ceiling* is the reference for any RL claim, not the unmitigated 6.6% baseline. Both denoised RL runs in RQ2 exceed this ceiling at peak (binary: 41.9%, dense: 39.6%) and stabilize above it in Q4 (binary: 35.2%, dense: 33.7%).

Table 4: $M{=}1$ **ablation.** *Naive* uses standard GRPO hyperparameters ($\eta{=}4{\times}10^{-5}$, $E{=}1$, $M{=}1$; Exp 9 of Table 8); *Scheme* rows fix our conservative configuration ($\eta{=}1{\times}10^{-6}$, $E{=}3$) and vary M. Per-sample binary success at matched 20 GRPO steps and $K{=}32$. Conservative LR + multi-epoch GRPO prevent catastrophic collapse at any horizon; multi-sample denoising additionally lifts success and tightens step-to-step variance.

Setting	Step 1	Step 20	Dip	Q1	Q4
Naive	32.0%	—	−63%	—	—
Scheme, $M{=}1$	21.9%	34.4%	−14%	25.0%	30.6%
Scheme, $M{=}5$	28.1%	**39.4%**	−11%	**31.6%**	**35.2%**

7.3 $M{=}1$ Ablation: Disentangling Denoising from Conservative Optimization

This ablation isolates the contribution of multi-sample averaging by holding $\eta = 1{\times}10^{-6}$ and $E{=}3$ fixed and setting $M{=}1$ (20 GRPO steps, 32 instructions per step, 640 total rollouts; results in Table 4).

Two findings. First, *stability comes from the optimization regime*: with $M{=}1$ the first-update dip is −14% relative, 4.5× smaller than the naive −63%, and the policy still trends upward across the full 20-step horizon. Second, *denoising delivers the absolute lift and tightens variance*: at matched optimization, $M{=}5$ gives +26.4% relative Q1 and +15.0% relative Q4 over $M{=}1$, and reduces the step-to-step spread by 38% (28.1 pp at $M{=}1$ vs. 17.5 pp at $M{=}5$). The catastrophic-collapse-prevention claim, therefore, belongs to the optimization regime, while the absolute-lift and

[2] Per-rep success rates: $35/128 = 27.3\%$ (seed 100) and $32/128 = 25.0\%$ (seed 101).

variance-reduction claims belong to denoising.

7.4 Binary vs Dense Reward Under the Scheme

The direct comparison between the denoised binary and denoised dense runs (Equation (10)) tests whether structured reward shaping is necessary once the reward estimator is denoised and the optimizer is conservative; the runs are matched on every other axis (K=32, M=5, η=1×10^{-6}, E=3, 20 steps).

Table 3 shows the two are indistinguishable: Q4 differs by 1.5 percentage points in favor of binary, well within step-to-step variance. Once M-sample averaging reduces downstream sampler noise and the conservative LR prevents overshooting, sparse binary feedback already carries enough between-instruction signal for GRPO. This is a positive practical result: binary reward is simpler to specify, free of shaping hyperparameters, and immune to the reward-hacking pitfalls.

8 Discussion

Why the Scheme Works The scheme addresses a structural property of two-model pipelines: a stochastic downstream sampler injects variance that can dominate the between-instruction variance the optimizer needs to exploit. The three components attack distinct aspects: multi-sample averaging (M=5) reduces estimator variance by a factor of M; the conservative learning rate (η=1×10^{-6}) bounds per-update KL drift; multi-epoch updates (E=3) amortize the $M\times$ rollout cost. Section 7.3 attributes collapse prevention to the optimization regime and the lift to denoising.

Reward-Hacking Risks Avoided The combined reward (Equation (10)) deliberately omits two components from our preliminary study (Appendix E) susceptible to reward hacking (Amodei et al., 2016; Gao et al., 2023; Coste et al., 2024): efficiency rewards (gameable by early termination) and stability reruns (redundant under group averaging). The retained static-bug penalty V_{bug} uses high-precision detectors based on abstract-syntax-tree (AST) analysis (Liang et al., 2023; Singh et al., 2022); false negatives are caught downstream by the binary success signal.

Generalization beyond robotics The reward-variance bottleneck in Section 6 is a property of any two-model pipeline whose upstream policy is optimized against a stochastic downstream sampler, not a specifically robotic phenomenon.

Reproducibility The instruction policy (Qwen3.5-35B-A3B with LoRA) and the code generator (Qwen3-Coder-Next) are open-weight; together with the published hyperparameters (Appendix A.6), the RQ2 and RQ3 trajectories are reproducible on commodity hardware, with the full set of experiments reported in this paper completing in roughly 160 GPU-hours on NVIDIA A100 80 GB cards.

9 Conclusion and Future Work

We presented Instruction-Conditioned Motion Control (ICMC), a three-stage framework (parameter specification, motion-instruction generation, code synthesis) with physics-based verification. With reasoning multimodal models, ICMC achieves 100% pick-and-place with o1, 60% with o4-mini (1.7× over one-shot), and a 3× lift on multi-step reshaping with o1 (Table 1).

To identify what drives these gains, we replaced the reasoning component with an open-weight text-only pipeline. Naive GRPO collapses (32%→11.7%); our scheme (M=5 denoising, LR 10^{-6}, E=3 epochs) lifts per-sample success 28.1%→39.4%, exceeding a 25.0% non-RL ceiling. RQ3 attributes collapse prevention to the optimization regime and the lift to denoising. The scheme trails the reasoning-model pipelines: ICMC's reliability comes primarily from unlocking reasoning multimodal models; the scheme is the best alternative when those are inaccessible.

Future Work (1) Increased statistical power via multi-seed runs at K=128, M=5, and 40–60 GRPO steps, plus held-out-prompt evaluation, to tighten the headline lifts beyond the current single-seed Q4 averages. (2) Physical-hardware validation under domain randomization with a task suite extended to pushing and insertion, using the present MuJoCo benchmark as a sim-to-real calibration anchor. (3) Joint optimization of π_θ and π_{code}, e.g. via alternating GRPO updates or a shared LoRA adapter, to test whether the reward-variance bottleneck weakens when the downstream sampler is itself adapting. (4) Multimodal scaling of the open-weight pipeline, layering visual observations and a vision–language backbone on top of the current text scene description, to extend the variance-control scheme to perceptually grounded settings.

Limitations

Three scoping decisions delineate our findings. First, MuJoCo offers a controlled setting for the variance analysis; physical-robot evaluation under sensor noise is a natural follow-up. Second, the RQ2/RQ3 ablation is deliberately text-only to isolate the decomposition and scheme from multimodal-grounding gains; layering visual conditioning on top is a natural next step for perceptually grounded tasks. Third, the o1 conditions in Table 1 use n=5 trials per cell with the o4-mini n=20 conditions as a variance reference; expanding to higher run counts and newer reasoning models would tighten the headline lifts.

Ethics Statement

Experimental scope All experiments in this paper are conducted in the MuJoCo physics simulator with no human subjects, no human-collected demonstration data, and no deployment on physical robots. The benchmark targets (12 letter shapes on a 7×7 anchor grid) and evaluation rollouts contain no personally identifiable, sensitive, or private information.

Artifact licenses and intended use The instruction policy and code generator are open-source Qwen models accessed under their respective licenses and used solely for research. The Qwen models are released under the Tongyi Qianwen license; MuJoCo (Apache 2.0), Gymnasium (MIT), TRL (Apache 2.0), and PEFT (Apache 2.0) are released under their respective open-source licenses; the OpenAI API models (o1, o4-mini, GPT-4o) are accessed under OpenAI's API terms of use. All artifacts used in this paper are applied within the scope of their intended use (physics simulation, RL training, parameter-efficient fine-tuning, language-model inference); the ICMC framework code, the 12-letter shape benchmark, and any trained LoRA adapters released alongside this paper are intended for research use, consistent with the access conditions of the underlying Qwen models.

Risk assessment We see no foreseeable harm or dual-use risk arising directly from the methods reported here. Broader risks of language-model-driven robot control, including unsafe autonomous behavior under distribution shift, calibration errors, and accountability for deployed systems, apply to the field at large rather than to this work specifically, and any sim-to-real follow-up should be paired with appropriate hardware safety reviews, formal task constraints, and human oversight.

Use of AI Assistance

AI-based writing assistants (large language models) were used for language polishing, paraphrasing of author-drafted notes, and grammar checks throughout the manuscript, consistent with ACL's policy on permitted uses of generative AI. All research questions, experimental design, methodology, implementation, data collection, analysis, figures, tables, and citations are the authors'; the authors verified every technical claim, equation, and reference in the paper and take full responsibility for its content.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, and 1 others. 2022. Do as i can, not as i say: Grounding language in robotic affordances.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*.

Dhruv Batra, Angel X Chang, Sonia Chernova, Andrew J Davison, Jia Deng, Vladlen Koltun, Sergey Levine, Jitendra Malik, Igor Mordatch, Roozbeh Mottaghi, and 1 others. 2020. Rearrangement: A challenge for embodied ai. *arXiv preprint arXiv:2011.01975*.

Homanga Bharadhwaj, Jay Vakil, Mohit Sharma, Abhinav Gupta, Shubham Tulsiani, and Vikash Kumar. 2024. Roboagent: Generalization and efficiency in robot manipulation via semantic augmentations and action chunking. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4788–4795. IEEE.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, and 1 others. 2022. Rt-1: Robotics transformer for real-world control at scale.

Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. 2024. Reward model ensembles help mitigate overoptimization. In *International Conference on Learning Representations*, volume 2024, pages 50905–50931.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, and 1 others. 2023. Palm-e: An embodied multimodal language model.

Leo Gao, John Schulman, and Jacob Hilton. 2023. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR.

Nikolaos Gkanatsios, Ayush Jain, Zhou Xian, Yunchu Zhang, Christopher Atkeson, and Katerina Fragkiadaki. 2023. Energy-based models are zero-shot planners for compositional scene rearrangement.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. volume 1, page 3.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. 2023. Voxposer: Composable 3d value maps for robotic manipulation with language models.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, and 1 others. 2022. Inner monologue: Embodied reasoning through planning with language models.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, and 1 others. 2024. Openvla: An open-source vision-language-action model.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. 2023. Reward design with language models.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. 2024. Chain of code: Reasoning with a language model-augmented code emulator. In *International Conference on Machine Learning*. PMLR.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pages 9493–9500. IEEE.

Weiyu Liu, Yilun Du, Tucker Hermans, Sonia Chernova, and Chris Paxton. 2022a. Structdiffusion: Language-guided creation of physically-valid structures using unseen objects.

Weiyu Liu, Chris Paxton, Tucker Hermans, and Dieter Fox. 2022b. Structformer: Learning spatial structure for language-guided semantic rearrangement of novel objects. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 6322–6329. IEEE.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, and 1 others. 2024. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pages 26516–26560.

Thinking Machines. 2026. Tinker: a training api for researchers and developers. https://tinker-docs.thinkingmachines.ai.

Oier Mees, Lukas Hermann, Erick Rosete-Beas, and Wolfram Burgard. 2022. Calvin: A benchmark for language-conditioned policy learning for long-horizon robot manipulation tasks. *IEEE Robotics and Automation Letters*, 7(3):7327–7334.

OpenAI. 2024. Openai o1 system card.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. volume 35, pages 27730–27744.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, and 1 others. 2022. A generalist agent. *arXiv preprint arXiv:2205.06175*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Mohit Shridhar, Lucas Manuelli, and Dieter Fox. 2022. Cliport: What and where pathways for robotic manipulation. In *Conference on robot learning*, pages 894–906. PMLR.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. 2022. Progprompt: Generating situated robot task plans using large language models.

Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. 2023. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2998–3009.

Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE.

Mark Towers, Ariel Kwiatkowski, John Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Kallinteris Andreas, Markus Krimmel, Arjun Kg, Rodrigo Perez-Vicente, and 1 others. 2026. Gymnasium: A standard interface for reinforcement learning environments. volume 38.

Sai H Vemprala, Rogerio Bonatti, Arthur Bucker, and Ashish Kapoor. 2024. Chatgpt for robotics: Design principles and model abilities. *Ieee Access*, 12:55682–55696.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformers Reinforcement Learning.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Andy Zeng, Pete Florence, Jonathan Tompson, Stefan Welker, Jonathan Chien, Maria Attarian, Travis Armstrong, Ivan Krasin, Dan Duong, Vikas Sindhwani, and 1 others. 2021. Transporter networks: Rearranging the visual world for robotic manipulation. In *Conference on Robot Learning*, pages 726–747. PMLR.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, and 1 others. 2023. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR.

A Setup, Implementation, and Compute

This appendix collects the shared MuJoCo/UR5e setup used by all three RQs, the RQ2/RQ3 RL implementation details (training configuration, training loop, reward components, and static-bug detectors), the per-configuration compute budget for the nine naive-GRPO baselines reported in Section 7.1, and a representative sample execution.

A.1 Simulation Environment

Figure 3 shows the MuJoCo simulation environment used for training and evaluation. The environment features a UR5e 6-DOF robot arm with a parallel gripper, positioned on a table workspace containing red cubes to be manipulated.

A.2 Robot API Specification

The robot control code uses the following API primitives provided by the MuJoCo-based Gymnasium environment:

```
# Environment initialization
env = gymnasium.make('manipulator_mujoco/TestEnv-v0',
                     render_mode='rgb_array')
observation, info = env.reset(seed=42)

# Object perception
pos = env.unwrapped.getCubePose(cube_id)  # Returns [x, y, z]

# Gripper control
env.unwrapped.closeGripper(0.001)  # Returns True when closed
env.unwrapped.openGripper(0.001)   # Returns True when opened

# Motion execution
trajectory = np.array([x, y, z, qx, qy, qz, qw])
obs, reward, done, trunc, info = env.step(trajectory)
```

A.3 Motion Instruction State Machine

The generated motion instructions follow a three-phase state machine:

1. **PERCEPTION:** Detect cube positions, get end-effector pose
2. **FETCHING:** Four sub-steps: move above cube, descend, close gripper, lift
3. **PLACING:** Two sub-steps: move to disposal area, open gripper

A.4 ICMC Pipeline Pseudocode

Algorithm 1 formalizes the three-stage ICMC pipeline introduced in Section 3.2: parameter specification, motion-instruction generation, and code synthesis, followed by physics-based verification. The state machine above is the inner control structure invoked by the synthesized code in Stage 3.

A.5 Sample Execution Results

Successful pick-and-place rollouts under the ICMC pipeline emit a per-cube `Error = X` line for each object and a terminal `Success: All cubes correctly placed!` line; the runner parses these to populate the binary success signal used downstream (Appendix A.8). Per-cube errors on a typical successful o4-mini rollout fall well within the RQ1 tolerance $\varepsilon = 0.1\,\mathrm{m}$, with cubes coming to rest comfortably inside the tolerance margin rather than at its edge.

A.6 Training Configuration

The two main RL runs in Section 6 (denoised binary and denoised dense) share the same training configuration apart from the reward specification. Beyond the headline hyperparameters reported in Section 6 (K=32 instructions per step, M=5 code samples per instruction, E=3 update epochs, learning rate $\eta = 1 \times 10^{-6}$, 20 total GRPO steps on the pick-and-place task), both runs use a LoRA adapter

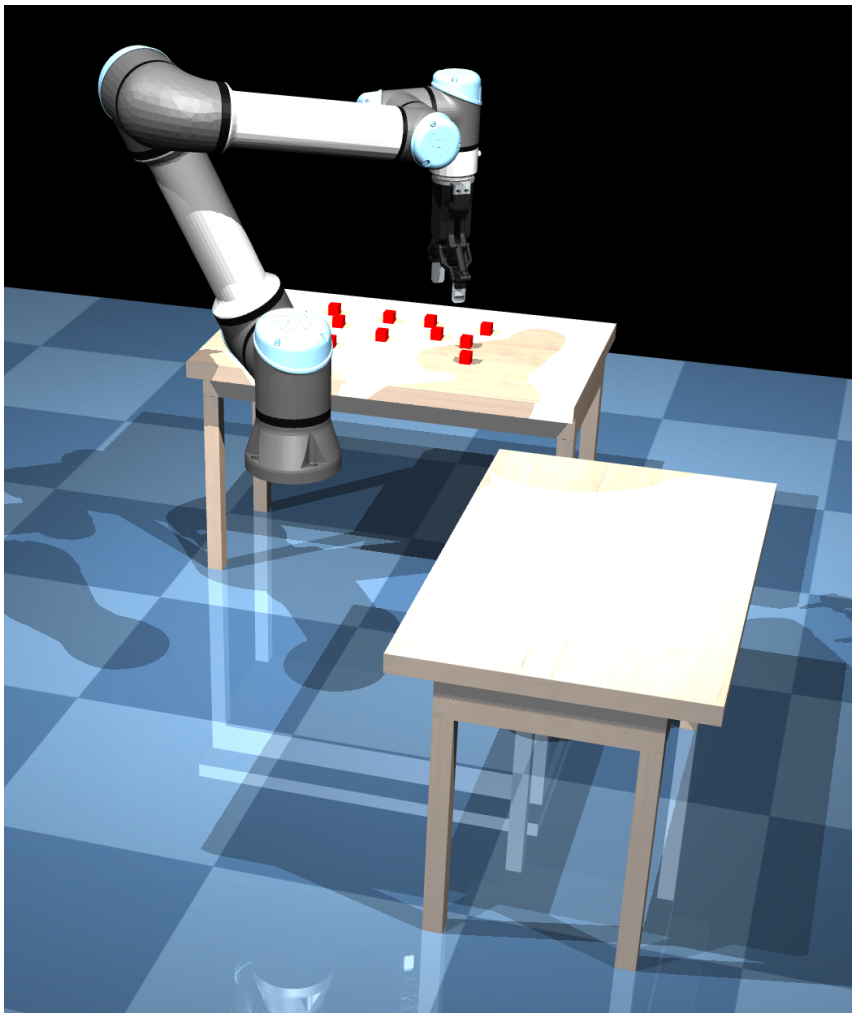

Figure 3: **MuJoCo Simulation Environment.** The pick-and-place task environment featuring a UR5e robot arm with parallel gripper. Red cubes on the table must be picked up and placed in the designated disposal area. The simulation provides RGB observations and physics-based execution for verifying generated motion control code.

of rank 32 and α=64 on the q/k/v/o projection matrices of the trainable Qwen3.5-35B-A3B instruction policy, with the frozen Qwen3-Coder-Next code generator served via OpenRouter. The GRPO update applies PPO-style clipping with ϵ_{clip}=0.2, KL-penalty coefficient β=0.01, gradient-norm clip 1.0, and group-level advantage normalization. Instruction sampling temperature is 0.7; downstream code sampling temperature is 0.1, kept low so that upstream-instruction-level variation dominates the reward variance studied in Section 7. Position tolerance is $\varepsilon = 0.05$ m, execution timeout is 120 s per rollout, and the maximum generation length is 4096 tokens. All RQ2/RQ3 runs use the prompt-fix addendum (Appendix A.12) and the raw (uncondensed) instruction format. The denoised-dense reward adds λ=0.3 on V_{bug}, α=0.25 on r_{goal}, and β=0.25 on r_{comp} to the binary baseline; the excluded efficiency and stability components are discussed in Appendix E. The preliminary study (Appendix E) used the Tinker API (Machines, 2026) with closed-source code generation; the main experiments instead run the instruction-policy GRPO loop locally with HuggingFace + LoRA against the OpenRouter-served frozen code generator.

Compute budget and infrastructure Across all RQ1, RQ2, and RQ3 experiments reported in this paper, the total GPU compute is approximately 160 GPU-hours on NVIDIA A100 80 GB cards (tensor-parallel TP=2 for both the trainable instruction policy and the frozen code generator served via vLLM). The total OpenAI API spend across the RQ1 reasoning-multimodal evaluations (o1 and o4-mini) is approximately 200 USD.

Algorithm 1 Instruction-Conditioned Motion Control (ICMC)

Input: Task description τ, visual observation o
Output: Task success $R(c, \tau) \in \{0, 1\}$

// Stage 1: Motion Control Parameter Specification
Define environment state, robot configuration, object IDs
Initialize execution variables: `steps`, `middle_step`, `break_flag`

// Stage 2: Motion Instruction Generation
$l \leftarrow \pi_{\text{instr}}(\tau, o)$ ▷ Generate structured instructions

// Stage 3: Motion Control Code Generation
$c \leftarrow \pi_{\text{code}}(\tau, o, l)$ ▷ Synthesize executable code

// Execution and Verification
Execute c in MuJoCo simulator
for each object i **do**
  Record final position p_i^{final}
end for
$R(c, \tau) \leftarrow \mathbf{1}[\max_i \|p_i^{\text{final}} - p_i^{\text{target}}\|_2 < \epsilon]$
Return: $R(c, \tau)$

Algorithm 2 ICMC with Denoised, Conservative GRPO

Input: Task distribution $\mathcal{D}$, instructions per step K, code samples per instruction M, learning rate η, GRPO epochs E
Initialize instruction policy π_θ from pretrained LLM
Fix code generator π_{code}
for each training step **do**
  Sample task τ from $\mathcal{D}$ (textual specification; no image in this pipeline)
  for $k = 1$ to K **do**
    Sample instruction $l_k \sim \pi_\theta(\cdot \mid \tau)$
    for $m = 1$ to M **do**
      Sample code $c_k^{(m)} \sim \pi_{\text{code}}(\cdot \mid \tau, l_k)$
      Execute in MuJoCo, obtain $r_k^{(m)}$ (binary or dense)
    end for
    Denoised reward: $\hat{r}_k \leftarrow \frac{1}{M} \sum_m r_k^{(m)}$
  end for
  Advantages: $A_k \leftarrow \hat{r}_k - \frac{1}{K} \sum_j \hat{r}_j$
  for $e = 1$ to E **do**
    Update θ by GRPO step on $\{(l_k, A_k)\}_{k=1}^K$ with learning rate η
  end for
end for
Return: Trained instruction policy π_θ

A.7 RL Training Algorithm

Per-token score function The policy-gradient term $\nabla_\theta \log \pi_\theta(l_k|s)$ in Section 6 factors as

$$\nabla_\theta \log \pi_\theta(l_k|s) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(l_k^{(t)}|s, l_k^{(<t)}), \tag{11}$$

the sum of per-token log-probability gradients over the T tokens of instruction l_k. When $A_k > 0$ (above-average reward) the update increases the probability of generating l_k; when $A_k < 0$ it decreases it; rollouts with $A_k = 0$ contribute no gradient and are dropped.

Per-step training loop The training loop is given as pseudocode in Algorithm 2. Per GRPO step:

1. **Task instance:** Sample a task specification τ from the task distribution (single-task pick-and-place in the main runs; no image observations in this text-only pipeline).

2. **Instruction sampling:** Draw K=32 instructions $l_k \sim \pi_\theta(\cdot \mid \tau)$ at temperature 0.7.

3. **Multi-sample code generation:** For each l_k, sample M=5 code candidates from π_{code} at temperature 0.1.

4. **Execution:** Run all $K \times M = 160$ code candidates in MuJoCo (timeout 120s).

5. **Denoised reward:** $\hat{r}_k = \frac{1}{M} \sum_m r_k^{(m)}$ following Equation (9), with $r_k^{(m)}$ either $\mathbb{1}[\text{success}_m]$ (binary reward) or the dense reward in Equation (10).

6. **Advantages:** $A_k = \hat{r}_k - \frac{1}{K} \sum_j \hat{r}_j$, then normalized.

7. **Multi-epoch update:** Apply E=3 GRPO gradient steps on the batch at $\eta = 1 \times 10^{-6}$.

The multi-epoch update ($E > 1$) amortizes the $M\times$ cost of denoising by extracting more gradient information per expensive rollout batch, following standard PPO practice (Schulman et al., 2017).

A.8 Task Success Verification

The binary success signal used in RQ2 is computed at the end of each rollout: for every target object i, we compute the Euclidean distance $\|p_i^{\text{final}} - p_i^{\text{target}}\|_2$ between its final XY position and its target, and the rollout succeeds iff all objects are within the position tolerance $\varepsilon = 0.05\,\text{m}$. In the deployed pipeline, the generated code prints a per-cube `Error = X` line for each object and a `Success: All cubes correctly placed` line on overall success; the runner parses these lines to populate the per-object position errors and the binary success flag. The cubes have a 3 cm edge, so the 5 cm tolerance ($\varepsilon = 0.05\,\text{m}$) admits a displacement of up to $5/3 = 1.\overline{6}$ edge lengths or 0.962 body diagonals ($\sqrt{3}\times 3 = 5.196\,\text{cm}$).

A.9 Static-Bug Detection (RQ2)

The three static-bug indicators (V_{sig}, V_{scope}, V_{map}), aggregated into V_{bug} for the RQ2 reward, are designed with a high-precision-over-high-recall principle. False positives (incorrectly penalizing valid code) harm learning by providing unfair negative signals; false negatives (missing actual bugs) are acceptable because execution failure ($r_{\text{success}} = 0$) serves as the ultimate safety net.

Signature Violation Detection (V_{sig}) We use AST parsing to detect modifications to predefined function signatures:

Algorithm 3 Signature violation detection (V_{sig}).

Input: candidate code c; expected signatures Σ
Output: (violated, $\mathcal{V}$)
Parse c into AST T; $\mathcal{V} \leftarrow [\,]$
for each `FunctionDef` node f in T **do**
  if f.name $\in \Sigma$ and params of $f \neq \Sigma[f.\text{name}]$ **then**
    append "Wrong params for f.name" to $\mathcal{V}$
  end if
end for
Return: $(|\mathcal{V}| > 0,\ \mathcal{V})$

AST parsing provides exact structural information, achieving 100% precision by construction: every reported violation corresponds to a literal AST parameter mismatch, with no fuzzy matching that could result in a false positive on valid code.

Scope Violation Detection (V_{scope}) We detect when global variables are assigned inside functions without a proper `global` declaration:

Algorithm 4 Scope violation detection (V_{scope}).

Input: candidate code c; expected globals G
Output: (violated, $\mathcal{V}$)
Parse c into AST T; $\mathcal{V} \leftarrow [\,]$
for each `FunctionDef` node f in T **do**
  $D \leftarrow$ names appearing in `Global` statements within f
  for each `Name` node n in f with n.id $\in G$ **do**
    if n appears in an assignment context (`Store`) and n.id $\notin D$ **then**
      append "Missing global: n.id" to $\mathcal{V}$
    end if
  end for
end for
Return: $(|\mathcal{V}| > 0,\ \mathcal{V})$

The key design decision is to only flag *assignments* (via AST's `Store` context), not reads. Reading global variables without declaration is valid Python; only assignment without `global` creates the scoping bug that causes silent failures.

Mapping Violation Detection (V_{map}) For pick-and-place tasks, we use regex on literal cube identifiers passed to the perception API to detect out-of-range indices:

Algorithm 5 Mapping violation detection (V_{map}) for pick-and-place.

Input: candidate code c; number of targets N
Output: (violated, $\mathcal{V}$)
$\mathcal{V} \leftarrow [\,]$
for each literal-argument call `getCubePose`(k) matched in c **do**
  if $k < 0$ or $k \geq N$ **then**
    append "Invalid cube_id: k" to $\mathcal{V}$
  end if
end for
Return: $(|\mathcal{V}| > 0,\ \mathcal{V})$

For reshaping tasks, an analogous detector flags out-of-range literal indices in `pattern_A[i]` / `pattern_B[i]` expressions. We intentionally only flag *literal out-of-range values*: regex cannot evaluate loop bounds or variable values, so flagging non-literal accesses would cause false positives on valid loop-based code such as `for i in range(num_targets): getCubePose(i)`.

The low recall of V_{map} is acceptable because undetected bugs still cause execution failure, resulting in $r_{\text{success}} = 0$. The static-bug penalties provide

Table 5: Design rationale for static-bug detection: prioritizing precision over recall.

Error Type	Definition	Impact on RL Training
False Positive	Flag correct code as buggy	Critical: unfairly penalizes good solutions
False Negative	Miss an actual bug	Acceptable: execution catches it ($r = 0$)

Table 6: Summary of static-bug detection capabilities.

Function	Method	Precision	Recall
V_{sig} (signature)	AST	100%	High
V_{scope} (scope)	AST	100%	Medium
V_{map} (mapping)	Regex	100%	Low

early feedback for common, obvious failure patterns, while the binary success signal remains the ultimate judge of code correctness.

A.10 Dense Reward Implementation

The denoised dense reward (Equation (10)) combines binary success with an aggregated static-bug penalty, a potential-based goal-progress reward, and a graded completion bonus. The per-sample implementation (invoked once per code sample, then averaged over M=5 samples per instruction) is:

Algorithm 6 Per-sample combined dense reward (binary + bug penalty + goal progress + completion bonus).

Input: execution result E with fields `success`, `init_distances`, `position_errors`; violation indicators $V_{\text{sig}}, V_{\text{scope}}, V_{\text{map}}$; weights λ, α, β; tolerance ε
Output: dense reward $r \in \mathbb{R}$
$r_{\text{binary}} \leftarrow \mathbf{1}[E.\text{success}]$
$V_{\text{bug}} \leftarrow (V_{\text{sig}} + V_{\text{scope}} + V_{\text{map}})/3$; $r_{\text{penalty}} \leftarrow \lambda \cdot V_{\text{bug}}$
// Goal-progress: normalized per-object distance improvement
for each object index i **do**
  $d_i \leftarrow E.\text{init_distances}[i]$; $d_f \leftarrow E.\text{position_errors}[i]$
  if $d_i > 0$ **then**
    $p_i \leftarrow \max\big(0,\ (d_i - d_f)/d_i\big)$
  else
    $p_i \leftarrow \mathbf{1}[d_f < \varepsilon]$
  end if
end for
$r_{\text{goal}} \leftarrow \text{mean}(p_1, p_2, \ldots)$ (0 if no objects)
// Completion bonus: graded terminal reward
if E.success **then**
  $r_{\text{comp}} \leftarrow 1.0$
else if $\max_i d_f^{(i)} < 2\varepsilon$ **then**
  $r_{\text{comp}} \leftarrow 0.5$
else
  $r_{\text{comp}} \leftarrow 0.0$
end if
Return: $r_{\text{binary}} - r_{\text{penalty}} + \alpha \cdot r_{\text{goal}} + \beta \cdot r_{\text{comp}}$

The three dense terms are gated as described in Section 7.4: the static-bug penalty only reduces the reward of structurally invalid code, while the goal-progress reward and completion bonus can credit near-miss rollouts that do not reach the strict tolerance $\varepsilon = 0.05$.

A.11 Naive GRPO Baseline Configurations

Table 7 summarizes per-configuration outcomes for the representative baselines discussed in Section 7.1, alongside the scheme for comparison. Table 8 then lists the per-configuration rollout count

for the full nine-baseline sweep. All runs use the upstream instruction policy Qwen3.5-35B-A3B with LoRA (rank 32) and the frozen open-weight code generator Qwen3-Coder-Next.

Error-category distribution Across the three condensed-instruction conditions (Exp 1 baseline, Exp 3 $K{=}32$, Exp 4 $K{=}128$; 3,840 rollouts total), the failure-mode distribution is essentially invariant to RL training: state-machine logic errors account for 44.7–47.8% of rollouts, `env.step` convention errors for 20.4–21.6%, and combined shape/index errors for 16.6–20.5%, confirming that the policy does not learn to fix the dominant failure modes under naive hyperparameters.

A.12 Prompt-Fix Addendum (Code-Gen)

The one-paragraph API-documentation addendum referenced in Section 7.2 addresses three common code-gen misunderstandings: (i) `env.step` expects a 2D trajectory of shape $(N, 8)$; (ii) the control loop must call manipulation functions repeatedly with `terminated` feedback; (iii) use `pattern_B` consistently throughout the generated code. The addendum is applied uniformly to all RQ2 and RQ3 runs and is the source of the $6.6\%{\to}25.0\%$ jump (no-RL baseline vs. prompt fix only in Table 7); it eliminates `env.step` convention errors entirely ($20.4\%{\to}0.0\%$).

A.13 $M{=}1$ Ablation: Trajectory and Variance Detail

The Conservative + $M{=}1$ run referenced in Section 7.3 trends upward across quarters as $25.0\%{\to}24.4\%{\to}25.0\%{\to}30.6\%$, with first-update dip -14% relative. At matched optimization settings ($\eta = 1{\times}10^{-6}$, $E{=}3$, $K{=}32$, 20 GRPO steps), $M{=}5$ binary delivers Step-1 = 28.1% vs. 21.9%, Q1 = 31.6% vs. 25.0% ($+26.4\%$ relative), Q4 = 35.2% vs. 30.6% ($+15.0\%$ relative), Step-20 = 39.4% vs. 34.4%.

The step-to-step spread is wider for $M{=}1$ (range 15.6%–43.8% across 20 steps; spread 28.1 pp) than for $M{=}5$ binary (range 24.4%–41.9%; spread 17.5 pp), consistent with the Monte Carlo variance argument in Section 6: the GRPO advantages under $M{=}5$ are estimates of the underlying $q(l)$ rather than single Bernoulli draws.

B RQ1 Experimental Details

This appendix collects the per-task experimental details for RQ1: shaping coordinate generation, plan-given reshaping with the Hungarian algorithm, and plan-generation reshaping, comparing the LLM against the Hungarian baseline.

B.1 Shaping Task: Coordinate Generation Evaluation

The Shaping task evaluates the LLM's ability to generate valid 2D coordinates for forming specific shapes (e.g., the letter 'M'), testing spatial reasoning capabilities independent of motion control execution. We measure two components separately so that spatial-reasoning errors and motion-control errors can be attributed cleanly: (i) coordinate accuracy

$$\mathrm{Acc}_{\mathrm{coord}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[C_i \subset W]\,, \qquad (12)$$

the fraction of runs in which every generated coordinate C_i lies inside the workspace rectangle $W = (0.4, 0.6) \times (-0.2, 0)$; and (ii) pick-and-place accuracy on the generated targets, which reduces to the pick-and-place rows of Table 1.

Setup o1-2024-12-17 with $N = 5$ independent trials per shape; the prompt asks for ≥ 5 coordinate points with $x \in (0.4, 0.6)$ and $y \in (-0.2, 0)$ that form the target letter. The o1 model achieved 100% coordinate accuracy (5/5 runs), generating valid M-shape coordinates within bounds in every trial.

B.2 Plan-Given Reshaping: Hungarian Algorithm Implementation

The planar reshaping task uses the Hungarian Algorithm to compute the optimal transformation between two patterns. This provides the LLM with a pre-computed plan specifying which cubes to remove, add, and remap.

Core Algorithm The transformation plan is computed as follows: pad both patterns to size $\max(|A|, |B|)$ with dummy nodes, build an Euclidean cost matrix between all source-target pairs, solve the assignment problem with the Hungarian algorithm, then classify each matched pair into `remove_idx` (real source to dummy target), `add_idx` (dummy source to real target), or `mapping` (real to real).

Execution Functions The plan drives a three-step state machine: `formulate_a_pattern(env, pattern_A, mapping)` builds the initial A configuration, `dispose_dummy_cubes(env, A_map,`

Table 7: **Naive GRPO vs. the scheme.** Representative configurations from the nine-baseline sweep at standard hyperparameters (η=4×10^{-5}, E=1, M=1, K=128 unless noted; full sweep including the raw-instruction conditions in Table 8). Naive RL stays on the prompt-fix-only ceiling without trending; only our scheme (last rows) trends upward.

Configuration	Per-sample success
Naive GRPO ($\eta = 4\times10^{-5}$, $E = 1$, $M = 1$):	
Baseline (no RL)	6.6%
Binary, K=128, 20 steps	flat (range 2.3–10.2%)
Prompt fix only (no RL)	25.0%
Binary + prompt fix, 20 steps	flat (range 21.1–45.3%)
Dense + prompt fix, 20 steps	flat (range 21.1–43.8%)
Our scheme (M=5, $\eta = 1\times10^{-6}$, $E = 3$; Table 3):	
Denoised binary	**upward** 28.1%→39.4%, **Q4** 35.2% (Figure 2)
Denoised dense	**upward** 29.7%→38.0%, **Q4** 33.7% (Figure 2)

Table 8: Per-configuration rollout count for the nine naive-GRPO baselines (Section 7.1). Total: 12,800 rollouts.

#	Configuration	Rollouts
1	Baseline (no RL)	640
2	Binary, K=128, 5 steps	640
3	Binary, K=32, 20 steps	640
4	Binary, K=128, 20 steps	2,560
5	Prompt fix (no RL)	640
7	Binary + prompt fix, 20 steps	2,560
8	Dense + prompt fix, 20 steps	2,560
9	Binary + raw, 10 steps	1,280
10	Dense + raw, 10 steps	1,280
Total		12,800

`remove_idx`) removes the cubes that do not appear in B, and `execute_remap(env, pattern_B, mapping, add_idx)` relocates the remaining cubes to their target positions and inserts new cubes at `add_idx` positions.

Example Transformation (M → I) Given source pattern M (5 nodes) and target pattern I (2 nodes), the Hungarian Algorithm produces the optimal plan:

- `remove_idx = [2, 3, 4]`: 3 cubes to dispose
- `add_idx = []`: no new positions needed
- `mapping = [[0, 0], [1, 1]]`: 2 cubes remap to form the vertical line

This Hungarian output is the "ground truth" row of Tables 9 and 10 below, against which the o1 and o4-mini outputs are compared.

B.3 Plan-Given Reshaping: o4-mini Failure Modes

The gap between 75% code validity and 15% reshaping success in o4-mini (Table 2) stems from runtime failures rather than syntactic bugs. The ICMC framework provides three predefined sub-task functions (`formulate_a_pattern`, `dispose_dummy_cubes`, `execute_remap`) that implement the state-machine logic; o4-mini frequently modifies these in ways that produce syntactically valid code but fail at runtime due to Python scoping rules and the non-standard state-machine pattern. The dominant categories observed are:

1. **Omitted globals.** Required variables (e.g., `pos`) are not declared with `global` inside the modified function, causing `UnboundLocalError` when the state machine resumes across function calls.

2. **Extraneous parameters.** The model passes `terminated` as a function argument instead of using the global variable updated by `env.step()`, breaking the termination contract.

3. **Mapping misuse.** The structured `mapping` list is iterated incorrectly (e.g., as a flat list rather than pairs of indices), causing index errors during remap execution.

The two improvements reported in Table 2 (amended sub-task functions, which correct the global declarations and enforce the proper signatures; and a constrained code-generation prompt, which instructs the model to preserve the predefined function signatures unless modification is strictly required) directly address (1) and (2). Mapping misuse (3) is addressed by both fixes together: the constrained prompt removes most rewrites, and the amended functions catch residual cases.

B.4 Plan-Generation Reshaping: LLM vs Hungarian Algorithm

This experiment evaluates whether the o1 model can replace the Hungarian Algorithm for computing optimal transformation plans between two patterns.

Experimental Setup

- **Model:** o1-2024-12-17
- **Number of runs:** 5 independent trials
- **Source pattern (M):** 5 nodes at coordinates shown below
- **Target pattern (I):** 2 nodes forming a vertical line

Pattern Coordinates The 5-node Pattern M and 2-node Pattern I used for the plan-generation evaluation correspond to the following layouts. M corresponds to the corner-and-middle layout

$$M = \{(0.41, -0.001),\ (0.41, -0.199),\ (0.5, -0.001),\ (0.59, -0.199),\ (0.59, -0.001)\},$$

and I is $\{(0.4, -0.1),\ (0.6, -0.1)\}$.

Prompt The model receives the two coordinate sets and is asked to return a Python dictionary with three lists: `remove_idx` (0-indexed source nodes to dispose), `add_idx` (0-indexed target positions needing new cubes), and `mapping` ($[i, j]$ pairs mapping $A[i]$ to $B[j]$). The prompt enforces three constraints: each source cube is either removed or mapped to exactly one target; each target position receives a mapped cube or appears in `add_idx`; the mapping minimizes total Euclidean distance. Cardinality consistency $|\text{remove_idx}| + |\text{mapping}| = |A|$ and $|\text{add_idx}| + |\text{mapping}| = |B|$ is also enforced.

Evaluation Criteria Each plan is assessed on validity (satisfies all constraints: correct list sizes, no duplicates, valid indices), correctness of the remove/add sets, and optimality of the total movement distance compared to the Hungarian baseline.

Results Table 9 reports the o1 plan-generation outputs across 5 runs.

o1 achieves an 80% success rate (4/5 valid plans), with all successful runs identical and differing from the Hungarian assignment only in the choice $A[3] \to B[1]$ vs $A[1] \to B[1]$ (both satisfy the constraints). Run 1 failed by removing all source cubes and adding new ones rather than reusing existing cubes, suggesting the model sometimes misreads the optimization objective.

Table 9: Plan-generation outputs from o1 across 5 runs on the M→I task. Run 1 failed by removing all source cubes; runs 2–5 produced an identical valid plan that differs from the Hungarian assignment but satisfies all constraints.

Run	remove_idx	mapping	Status
1	[0,1,2,3,4]	[]	FAIL (removes all)
2	[1,2,4]	[[0,0], [3,1]]	Valid
3	[1,2,4]	[[0,0], [3,1]]	Valid
4	[1,2,4]	[[0,0], [3,1]]	Valid
5	[1,2,4]	[[0,0], [3,1]]	Valid
Hungarian (ground truth)			
–	[2,3,4]	[[0,0], [1,1]]	Optimal

Table 10: Plan-generation outputs from o4-mini across 5 runs on the M→I task. All runs produced valid plans, with three distinct mapping strategies; total movement distance is 0.199 vs. the Hungarian baseline's 0.314.

Run	remove_idx	mapping	Status
1	[1,2,3]	[[0,0], [4,1]]	Valid
2	[0,2,4]	[[1,0], [3,1]]	Valid
3	[1,2,4]	[[0,0], [3,1]]	Valid
4	[1,2,4]	[[0,0], [3,1]]	Valid
5	[1,2,3]	[[0,0], [4,1]]	Valid

o4-mini Results Table 10 reports the o4-mini plan-generation outputs across 5 runs.

o4-mini achieves a 100% success rate (5/5 valid plans) with three distinct mapping strategies, all of which achieve shorter total movement distance (0.199 units) than the Hungarian baseline (0.314 units). This demonstrates that multiple optimal solutions exist for this pair, and o4-mini outperforms o1 here on both validity (100% vs 80%) and movement efficiency.

C Benchmark Details

This section provides complete details of the character shaping and reshaping benchmark, including shape definitions, difficulty metrics, and reshaping plan statistics.

C.1 Benchmark Design Justification

We select 12 character targets as a structurally representative suite rather than enumerating all 26 letters. Table 11 shows how these 12 shapes span the primary geometric and constraint classes that govern rearrangement difficulty.

Table 11: Structural motif families covered by the 12-shape benchmark. Each family tests distinct manipulation capabilities.

Motif Family	Shapes	Manipulation Challenge
Linear stroke	I, L	Alignment, endpoint accuracy
Junction/branching	T, E, F, H	Multi-contact placement, non-local constraints
Diagonal structure	N, Z, X	Spatial coupling, breaks greedy heuristics
Enclosure topology	O	Boundary formation, "closing the loop"
Global symmetry	M	Coordinated bilateral placement

The remaining 14 letters (A, B, C, D, G, J, K, P, Q, R, S, V, W, Y) are combinations of these same motifs: C/D/G/Q are "O-like but open" (enclosure family), V/W/Y belong to the diagonal/symmetry family, K/R combine branching with diagonals, and A/P/B combine branching with partial enclosure. Going from 12 to 26 targets would mainly add surface variations without new manipulation phenomena, while reducing the number of trials per condition. Additionally, some letters (e.g., B, S) contain curved strokes that cannot be faithfully represented on a discrete grid, shifting the evaluation from manipulation capability to geometric approximation fidelity.

Why 66 reshaping pairs Given 12 shaping targets, the reshaping benchmark consists of all $\binom{12}{2} = 66$ unordered target pairs. This is not an arbitrary design choice but a mathematical consequence: 66 is the minimal complete set that exhausts all pairwise structural transformations among the chosen targets, including stroke↔stroke (e.g., I↔L), stroke↔junction (I↔T), junction↔diagonal (E↔N), diagonal↔diagonal (N↔Z), non-enclosure↔enclosure (U↔O), asymmetric↔symmetric (F↔M), and enclosure↔symmetric (O↔M). Any smaller subset would omit entire classes of structural transitions, biasing evaluation toward particular transformation families. By evaluating the full pairwise set, we avoid biases introduced by hand-selected or metric-dependent subsets and enable stratified analysis across transformation types.

Directionality as experimental condition We treat directionality (A→B vs B→A) as an experimental condition rather than distinct benchmark tasks, because both directions share the same underlying structural relationship. The benchmark definition uses 66 unordered pairs; directional asymmetries (e.g., moving *into* an enclosure vs *out of* one) can be analyzed as a secondary research question within this framework.

C.2 Shaping Tasks Visualization

Figure 4 visualizes all 12 character shaping tasks. Each subplot shows a target shape rendered on the 7×7 anchor grid, with cube counts ranging from 8 (L) to 13 (M).

C.3 Shape Coordinate Definitions

Each character shape is defined as a set of grid coordinates (c, r) on a 7×7 anchor grid. The mapping to workspace coordinates is:

$$x(c) = 0.405 + 0.031c, \quad c \in \{0, \ldots, 6\} \quad (13)$$

$$y(r) = -0.195 + 0.031r, \quad r \in \{0, \ldots, 6\} \quad (14)$$

The coefficients are derived from the robot workspace constraints: the manipulation area spans $x \in (0.4, 0.6)$ and $y \in (-0.2, 0)$, giving a 0.2×0.2 m region. We fit seven anchor points with step size 0.031 m and offsets $(0.405, -0.195)$ that center the grid within the bounds; this yields anchor positions from $(0.405, -0.195)$ to $(0.591, -0.009)$, with 5 mm clearance to the lower bound and 9 mm to the upper bound on each axis. Table 12 provides the complete grid coordinate definitions for all 12 character shapes.

C.4 Reshaping Difficulty Stratification

The composite difficulty metric from Equation (7) combines edit distance with topological penalties:

$$D(A, B) = \Delta(A, B) + 6\,\mathbf{1}_{\text{encl}} + 3\,\mathbf{1}_{\text{sym}} \quad (15)$$

where the symmetric shapes are I, T, H, X, O, M (bilateral symmetry about the vertical axis), and O is the only shape with enclosure topology. With 12 shapes, there are $\binom{12}{2} = 66$ unordered reshaping pairs; Table 13 reports the resulting difficulty distribution.

Figure 5 visualizes representative reshaping examples from each difficulty level, showing source and target configurations side by side.

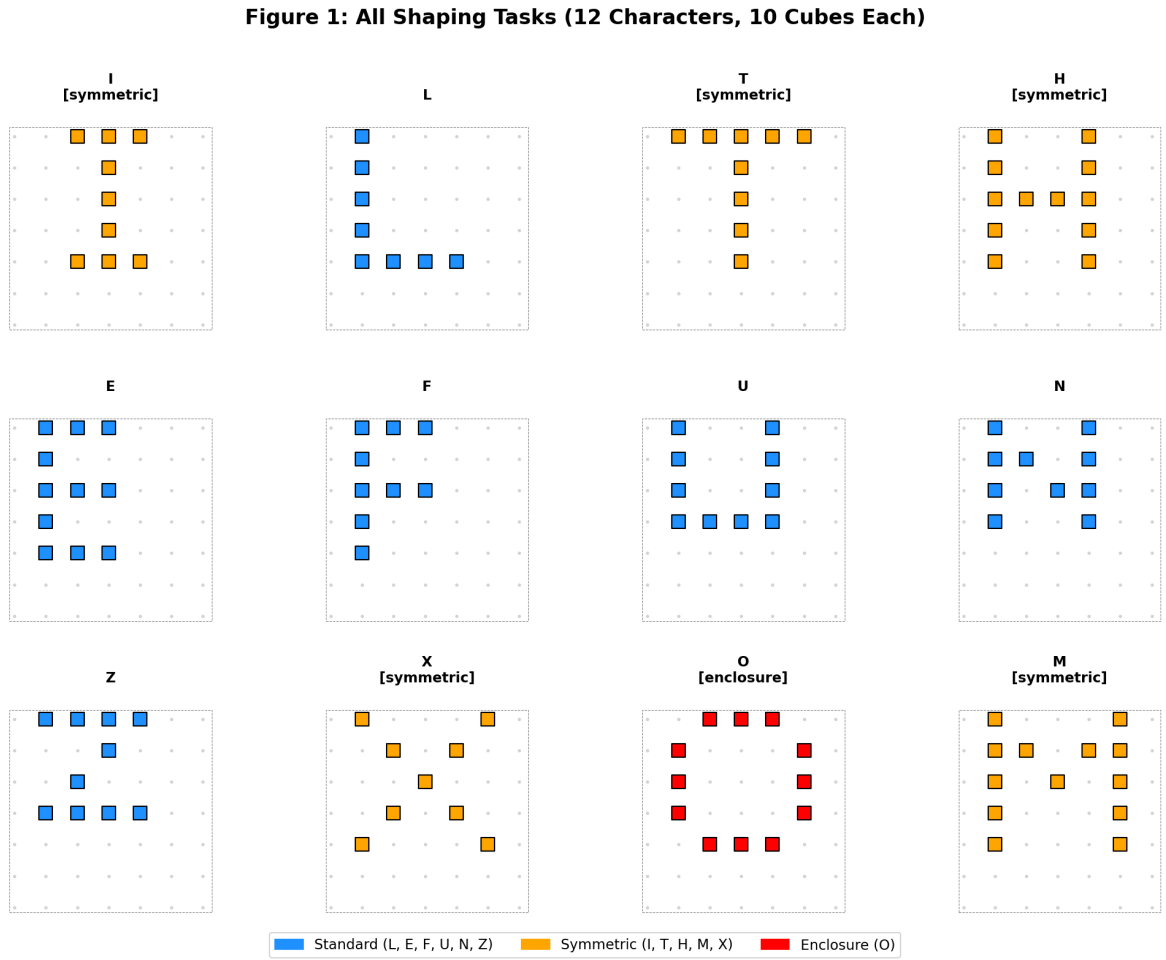

Figure 4: **All 12 Shaping Tasks.** Each character target is defined on a 7×7 anchor grid within workspace bounds $x \in (0.4, 0.6)$, $y \in (-0.2, 0)$. Shapes span diverse geometric properties: vertical strokes (I, L), horizontal elements (T, E, F), bilateral symmetry (H, M, X), diagonals (N, Z, X), and enclosure (O). Cube counts range from 8 to 13.

Figure 6 shows the full 12×12 difficulty matrix as a heatmap, where each cell (i, j) represents the composite difficulty D for reshaping from shape i to shape j.

C.5 Reshaping Plan Statistics

Using the Hungarian Algorithm to compute optimal transformation plans, we analyze the reshaping operations across all 132 ordered pairs (66 unordered pairs $\times$ 2 directions). The algorithm minimizes total cube movement distance while computing:

- n_{remove}: Cubes to remove (in source but not target)
- n_{add}: Positions to add cubes (in target but not source)
- n_{relocate}: Cubes that move to different positions (mapped but different coordinates)

Table 14 summarizes the reshaping plan statistics across the 132 ordered pairs.

Key Observations

- **Balanced operations:** $n_{\text{remove}} = n_{\text{add}}$ in expectation across the 132 ordered pairs, since each unordered pair contributes the same total but in opposite directions of $A \to B$ vs. $B \to A$.
- **Relocation dominates:** On average, 4.88 cubes are relocated per transformation, while only 0.88 are removed and 0.88 added; pairs with the same cube count (e.g. I↔T) trigger zero remove/add but still require relocation.
- **Distance correlation:** Total movement distance correlates with edit distance (Pearson $r = 0.567$), validating Δ as a coarse proxy for planning complexity.
- **Special case (E→F):** F is a subset of E (all 9 positions in F exist in E). The transformation

Table 12: Complete shape definitions as grid coordinates (c, r). Cube counts range from 8 (L) to 13 (M).

Shape	Cubes	Grid Coordinates (c, r)
I	9	(3,2), (3,3), (3,4), (3,5), (3,6), (2,6), (4,6), (2,2), (4,2)
L	8	(1,2), (1,3), (1,4), (1,5), (1,6), (2,2), (3,2), (4,2)
T	9	(1,6), (2,6), (3,6), (4,6), (5,6), (3,5), (3,4), (3,3), (3,2)
H	12	(1,2), (1,3), (1,4), (1,5), (1,6), (2,4), (3,4), (4,2), (4,3), (4,4), (4,5), (4,6)
E	11	(1,2), (1,3), (1,4), (1,5), (1,6), (2,6), (3,6), (2,4), (3,4), (2,2), (3,2)
F	9	(1,2), (1,3), (1,4), (1,5), (1,6), (2,6), (3,6), (2,4), (3,4)
U	10	(1,3), (1,4), (1,5), (1,6), (2,3), (4,3), (4,4), (4,5), (4,6), (3,3)
N	10	(4,3), (4,4), (4,5), (1,3), (1,4), (1,5), (1,6), (2,5), (3,4), (4,6)
Z	10	(1,6), (2,6), (3,6), (3,5), (2,4), (1,3), (2,3), (3,3), (4,3), (4,6)
X	9	(1,6), (2,5), (3,4), (4,3), (5,2), (5,6), (4,5), (2,3), (1,2)
O	12	(2,6), (3,6), (4,6), (2,2), (3,2), (4,2), (1,5), (1,4), (1,3), (5,5), (5,4), (5,3)
M	13	(3,4), (1,2), (1,3), (1,4), (1,5), (1,6), (5,2), (5,3), (5,4), (5,5), (5,6), (2,5), (4,5)

Table 13: Difficulty stratification of 66 reshaping pairs. Example pairs are shown for each level.

Level	D Range	Count	Example Pairs
Easy	$D \leq 8$	8	E↔F, I↔T, L↔F
Medium	$9 \leq D \leq 14$	28	H↔M, F↔N, E↔Z
Hard	$15 \leq D \leq 19$	22	I↔O, T↔O, L↔O
Extreme	$D \geq 20$	8	H↔O, F↔O, U↔O

Table 14: Reshaping plan statistics across all 132 ordered pairs (mean ± std).

Metric	Mean	Std	Min	Max
n_{remove}	0.88	1.25	0	5
n_{add}	0.88	1.25	0	5
n_{relocate}	4.88	1.84	0	9
Total Distance	0.262	0.097	0.000	0.479
Δ (Edit Distance)	11.12	3.63	2	21
D (Composite)	13.76	4.87	2	27

requires removing 2 cubes with zero movement (Total Distance $= 0$, $\Delta = 2$).

Figure 7 visualizes the optimal reshaping plans computed by the Hungarian Algorithm for representative pairs. The visualization shows source positions (blue/cyan), target positions (green/yellow), cubes to remove (red), and positions to add (orange).

D Difficulty Metric Coefficient Validation

This section provides empirical validation for the choice of coefficients $(\alpha, \beta, \gamma) = (1, 6, 3)$ in the composite difficulty metric $D(A, B)$.

D.1 Ground Truth Metrics

We define the ground-truth manipulation complexity using the operation count computed via Hungarian algorithm optimal assignment. Given source pattern A and target pattern B, the Hungarian algorithm computes the minimum-cost bipartite matching to determine:

- n_{remove}: Number of cubes to remove (present in A but not in B)
- n_{add}: Number of positions to add cubes (present in B but not in A)
- n_{remap}: Number of cubes that move to different positions (mapped but with nonzero displacement)

The total operation count is defined as:

$$n_{\text{ops}} = n_{\text{remove}} + n_{\text{add}} + 2 \cdot n_{\text{remap}} \tag{16}$$

where the factor of 2 for remap operations accounts for both pick-up and place-down actions. Remove and add operations each count as 1 because they involve a single placement action (dispose to free area, or fetch from free area), while remap requires the complete pick-and-place sequence.

D.2 Evaluation Methodology

We evaluate 10 coefficient combinations across 16 reshaping pairs stratified by difficulty (4 pairs per level: Easy, Medium, Hard, Extreme). For each coefficient set (α, β, γ), we compute $D(A, B)$ for all pairs and measure Spearman rank correlation with the ground-truth operation count.

Table 15 shows the ground truth data for all 16 test pairs.

D.3 Coefficient Evaluation Results

Table 16 presents the Spearman rank correlation between $D(A, B)$ and ground-truth metrics for each coefficient combination.

Table 15: Ground truth data for 16 test pairs. Δ = symmetric difference, enc = enclosure flip, sym = symmetry flip, n_{ops} = operation count per Equation (16).

Level	Pair	Δ	enc	sym	n_{rem}	n_{add}	n_{remap}	n_{ops}
Easy	E→F	2	0	0	2	0	0	2
	I→T	4	0	0	0	0	2	4
	U→N	4	0	0	0	0	2	4
	L→E	5	0	0	0	3	1	5
Medium	H→U	6	0	1	2	0	2	6
	F→N	9	0	0	0	1	5	11
	F→Z	9	0	0	0	1	4	9
	L→U	10	0	0	0	2	4	10
Hard	I→H	15	0	0	0	3	6	15
	I→F	12	0	1	0	0	6	12
	I→O	9	1	0	0	3	3	9
	T→H	15	0	0	0	3	6	15
Extreme	M→I	20	0	0	4	0	8	20
	H→O	14	1	0	0	0	7	14
	F→O	11	1	1	0	3	6	15
	Z→M	19	0	1	0	3	8	19

Table 16: Coefficient evaluation results. $\rho(n_{\text{ops}})$ = Spearman correlation with operation count; $\rho(\text{dist})$ = Spearman correlation with Hungarian distance.

Coefficients (α, β, γ)	$\rho(n_{\text{ops}})$	$\rho(\text{dist})$	Note
(1, 0, 0)	0.975	0.840	Δ only
(1, 0, 3)	0.960	0.844	Δ + sym
(1, 3, 0)	0.957	0.785	Δ + small enc
(1, 3, 3)	0.949	0.769	Equal small
(1, 6, 3)	**0.919**	**0.722**	**Proposed**
(1, 6, 0)	0.915	0.715	Δ + enc
(1, 6, 6)	0.889	0.701	Equal medium
(1, 9, 3)	0.859	0.641	High enc
(1, 9, 6)	0.849	0.640	Both high
(1, 12, 6)	0.803	0.588	Very high enc

D.4 Key Observations

All tested coefficient sets achieve Spearman $\rho > 0.80$ with operation count, indicating that the symmetric difference Δ is the dominant factor in predicting manipulation complexity. For the 4 easy pairs (E→F, I→T, U→N, L→E), which have no enclosure or symmetry flips, all coefficient sets achieve perfect rank correlation ($\rho = 1.0$): Δ alone suffices when topology is unchanged.

The remaining variation across coefficient sets comes from O-pairs and symmetry-flip pairs. Without topology penalties, the Δ-only metric $(1, 0, 0)$ ties cases that require qualitatively different operations: I→O and L→T both have $\Delta = 9$, but I→O requires forming a closed boundary (order-sensitive placement to avoid blocking the interior) while L→T is a translation-only edit with no topology change. The $(1, 6, 3)$ coefficients correctly up-rank the I→O case (giving $D = 9 + 6 = 15$, Hard) while leaving L→T at $D = 9$ (Medium); they sacrifice 5.6 percentage points of raw correlation against n_{ops} in exchange for this topological discrimination. The penalty magnitudes are calibrated to be on the same numeric scale as Δ (a $+6$ enclosure flip equals 6 unit edit operations; a $+3$ symmetry flip equals 3), ensuring the composite metric remains interpretable as edit-distance-plus-structural-cost.

E Preliminary Study: API-Based RL Training

This appendix presents preliminary RL training results obtained using cloud-based API models (Qwen3-4B via Tinker API for the instruction policy; o4-mini via OpenAI API for frozen code generation). These experiments validated the staged training design and reward formulations before we transitioned to a fully local pipeline for the main experiments. The numbers in this section are archival: the underlying API logs are not part of the local experimental artifacts shipped with the paper, so the trajectories below should be read as historical context that motivates the scheme in Section 6, not as primary results.

E.1 Stage A: Binary RL with Static-Bug Penalties

Setup The instruction policy (Qwen3-4B, LoRA rank 32, lr 4×10^{-5}) was trained with GRPO-style

policy gradients using the Tinker API (Machines, 2026). The code generator (o4-mini) remained frozen. GPT-4o-mini served as a lightweight code repair agent for variable name mismatches.

Before RL, three targeted settings addressed dominant failure modes from RQ1:

1. **Task-specific prompt templates** specifying expected code structure and variable naming conventions.

2. **Constrained code generation instruction:** "*Please do not change the functions in [Basic operational codes] unless you must do that.*"

3. **Post-generation code repair** via GPT-4o-mini for surface-level syntactic inconsistencies.

With these settings, the untrained pipeline achieved 40% task success on both pick-and-place (pattern I) and reshaping (M→I) over 5 trials each.

Reward Binary success augmented with static-bug penalties:

$$r_k = \mathbb{1}[\text{task_success}] - 0.2 \cdot V_{\text{sig}} - 0.2 \cdot V_{\text{scope}} - 0.2 \cdot V_{\text{map}} \quad (17)$$

where V_{sig}, V_{scope}, and V_{map} are binary static-bug indicators computed via AST analysis (signature violations), scope analysis (missing `global` declarations), and regex matching (out-of-range indices), respectively.

Training $K = 5$ parallel rollouts per step, $E = 3$ GRPO epochs, 60:40 reshaping-to-pick-and-place task mix. Training ran for 50 steps.

Results Starting from 20% baseline, Stage A achieved 48% weighted success at step 50 (pick-and-place: 60%, reshaping: 40%), a 28-point absolute improvement. The learning curve exhibited non-monotonic behavior characteristic of RL under sparse rewards, with a dip at steps 30–40 before recovery at step 50.

E.2 Stage B: Dense Reward (Gated Bonus)

Setup Initialized from the Stage A step-50 checkpoint. More conservative hyperparameters: $E = 2$ epochs, lr 5×10^{-5}, tighter PPO clipping.

Reward Gated Bonus formulation: $R^{\text{gated}} = r_{\text{base}} \cdot (1 + 0.2 \cdot r_{\text{stab}})$, where $r_{\text{base}} = 0.35 \cdot r_{\text{goal}} + 0.45 \cdot r_{\text{comp}} + 0.20 \cdot r_{\text{eff}}$.

The four components are:

Table 17: Preliminary Stage B results (API-based, 50 steps). Both reward configurations improve over the Stage A baseline. Weighted = $0.4 \times$ P&P $+ 0.6 \times$ Reshaping.

Config	Step	P&P	Reshape	Weighted	Δ
Additive	10	40%	80%	64%	+16
Additive	30	100%	40%	64%	+16
Gated	10	20%	100%	**68%**	**+20**
Gated	30	0%	100%	60%	+12

- r_{goal} (Goal-progress): $\frac{1}{N}\sum_{i=1}^{N} \max(0, (d_i^{\text{init}} - d_i^{\text{final}})/d_i^{\text{init}})$, the normalized distance improvement per object.

- r_{comp} (Completion): 1.0 if $e_{\text{final}} < \varepsilon$; 0.5 if $< 2\varepsilon$; 0 otherwise ($\varepsilon = 0.05$).

- r_{eff} (Efficiency): $p \cdot \exp(-T_{\text{exec}}/60)$, gated by success rate p and valid execution time.

- r_{stab} (Stability): p^2 where p is success rate across $E = 2$ reruns of the same code.

We also tested an Additive variant: $R^{\text{add}} = 0.35 r_{\text{goal}} + 0.45 r_{\text{comp}} + 0.20 r_{\text{eff}} + 0.20 r_{\text{stab}}$.

Results (50 steps) Both configurations improved over the Stage A baseline (48%):

The Gated Bonus achieved the highest peak: 68% weighted success at step 10 (+20 over Stage A). However, both configurations exhibited non-monotonic convergence, with later checkpoints degrading, likely due to reward proxy overfitting under the small evaluation sample ($n = 5$ per task). The Gated configuration showed schema overfitting: 100% reshaping success but 0–20% pick-and-place, attributable to the multiplicative stability bonus amplifying gradients for complex reshaping patterns.

E.3 Lessons Carried Forward to the Main Scheme

The preliminary study informed several design choices in Sections 6 and 7, both positively (kept) and negatively (discarded):

1. **Reasoning code generators hide the reward-variance problem.** The preliminary study used o4-mini, a strong reasoning code generator that follows instructions deterministically; this masked the downstream-sampler variance that dominates when the code generator is an open-weight model (Qwen3-Coder-Next), as

documented in Section 7.1. The variance argument behind the scheme in Section 6 was identified once we transitioned to open-weight code generation.

2. **Staged training was discarded.** Although the preliminary 20% → 48% → 68% trajectory looked encouraging, the Stage B curves were non-monotonic and the Gated Bonus exhibited schema overfitting (100% reshaping vs. 0–20% pick-and-place at later checkpoints), symptoms of reward-proxy overoptimization. The main experiments instead train two independent models from scratch under denoised binary and denoised dense rewards (Section 6), enabling a clean controlled comparison.

3. **Efficiency and stability terms were dropped.** Efficiency rewards proved gameable (the policy learned to terminate early), and stability reruns are redundant once the GRPO group provides $K \times M$ samples. The dense reward in Equation (10) retains only goal-progress, completion, and static-bug penalty.

4. **Static-bug penalties were retained** but with λ=0.3 on the aggregated V_{bug} rather than separate 0.2 weights per detector, since the AST-based detectors (signature, scope, mapping) target a single code-quality dimension.

5. **Transition to an open-weight pipeline.** The closed-source preliminary study made multi-seed, longer-horizon experiments difficult to scale. The current scheme runs against an open-weight code generator served via OpenRouter, enabling the longer-horizon, multi-seed setting used in the main experiments.

F Wilson 95% Confidence Intervals for Tables 1 and 2

This appendix reports Wilson 95% confidence intervals for every binomial cell in Table 1 (Main Results) and Table 2 (Reshaping ablation + outside baselines). Wilson score is the standard closed-form 95% CI for a binomial proportion; it is better behaved than the normal-approximation interval at small n or at p near 0/1 and does not require bootstrap simulation.

F.1 Table 1 (Main Results) cells

Sample sizes per the Table 1 caption: o1 = 5 runs, o4-mini = 20 runs. Shaping ($\text{Acc}_{\text{coord}}$) is a coordinate-accuracy metric over multiple shapes rather than a single binomial cell and is omitted from the CI table.

Table 18: Wilson 95% CIs for Table 1 cells. Format: point % [lower, upper] (x/n).

Task	Model	One-shot	ICMC
Pick-and-Place	o1	100.0 [56.6, 100.0] (5/5)	100.0 [56.6, 100.0] (5/5)
Pick-and-Place	o4-mini	35.0 [18.1, 56.7] (7/20)	60.0 [38.7, 78.1] (12/20)
Reshaping	o1	20.0 [3.6, 62.4] (1/5)	60.0 [23.1, 88.2] (3/5)
Reshaping	o4-mini	25.0 [11.2, 46.9] (5/20)	15.0 [5.2, 36.0] (3/20)
Reshaping (plan-gen)	o1	80.0 [37.6, 96.4] (4/5)	
Reshaping (plan-gen)	o4-mini	100.0 [56.6, 100.0] (5/5)	

All four one-shot-vs-ICMC lift comparisons in Table 1 have overlapping per-cell CIs at current sample sizes: Pick-and-Place o1 [56.6, 100.0] vs [56.6, 100.0] (both saturated at 5/5), Pick-and-Place o4-mini [18.1, 56.7] vs [38.7, 78.1] (overlap region [38.7, 56.7]), Reshaping o1 [3.6, 62.4] vs [23.1, 88.2] (overlap [23.1, 62.4]), and Reshaping o4-mini [11.2, 46.9] vs [5.2, 36.0] (point estimates already favor one-shot, full overlap). The point estimates still motivate the qualitative claims in §RQ1, but the per-cell CI overlap confirms the §Limitations expansion-as-follow-up framing: a 4× expansion of the o1 cells (and a complementary increase on the extreme-bucket reshaping pair for o4-mini) would tighten these intervals and is the natural path to upgrading the lift claims from point estimates to statistically distinguishable effects.

F.2 Table 2 (Reshaping ablation + outside baselines) cells

All cells: $n = 20$ trials on the M→I pair ($D = 20$, Extreme bucket).

Table 19: Wilson 95% CIs for Table 2 cells. Format: point % [lower, upper] (x/n).

Configuration	Code Valid	Task Success
Chain-of-Code	55.0 [34.2, 74.2] (11/20)	20.0 [8.1, 41.6] (4/20)
Iterative correction	15.0 [5.2, 36.0] (3/20)	5.0 [0.9, 23.6] (1/20)
Baseline ICMC	75.0 [53.1, 88.8] (15/20)	15.0 [5.2, 36.0] (3/20)
+ Amended functions	100.0 [83.9, 100.0] (20/20)	20.0 [8.1, 41.6] (4/20)
+ Constrained prompt	70.0 [48.1, 85.5] (14/20)	35.0 [18.1, 56.7] (7/20)
+ Both	90.0 [69.9, 97.2] (18/20)	50.0 [29.9, 70.1] (10/20)

The Baseline ICMC → + Both ablation chain (Task Success 15% [5.2, 36.0] → 50% [29.9, 70.1]) does *not* overlap, so the targeted-fix lift in Table 2 is statistically clean at $n = 20$. The outside-baseline comparison Chain-of-Code (20% [8.1, 41.6]) vs +

Both (50% [29.9, 70.1]) also does not overlap, supporting the §RQ1 "ICMC's structural decomposition combined with task-specific fixes outperforms isolated prompting baselines" claim. The Iterative correction row (5% [0.9, 23.6]) is the weakest baseline and is statistically below the + Constrained prompt and + Both rows; it borderline overlaps with Baseline ICMC.

F.3 Paired bootstrap CIs for the lifts

A natural complement to the per-cell Wilson CIs is a paired bootstrap CI for each one-shot→ICMC lift, using the same task pairs across both conditions. This requires the per-trial paired data (which seed produced which outcome under each method), and is deferred to a future revision pending the expanded o1 evaluation flagged in the Limitations section.

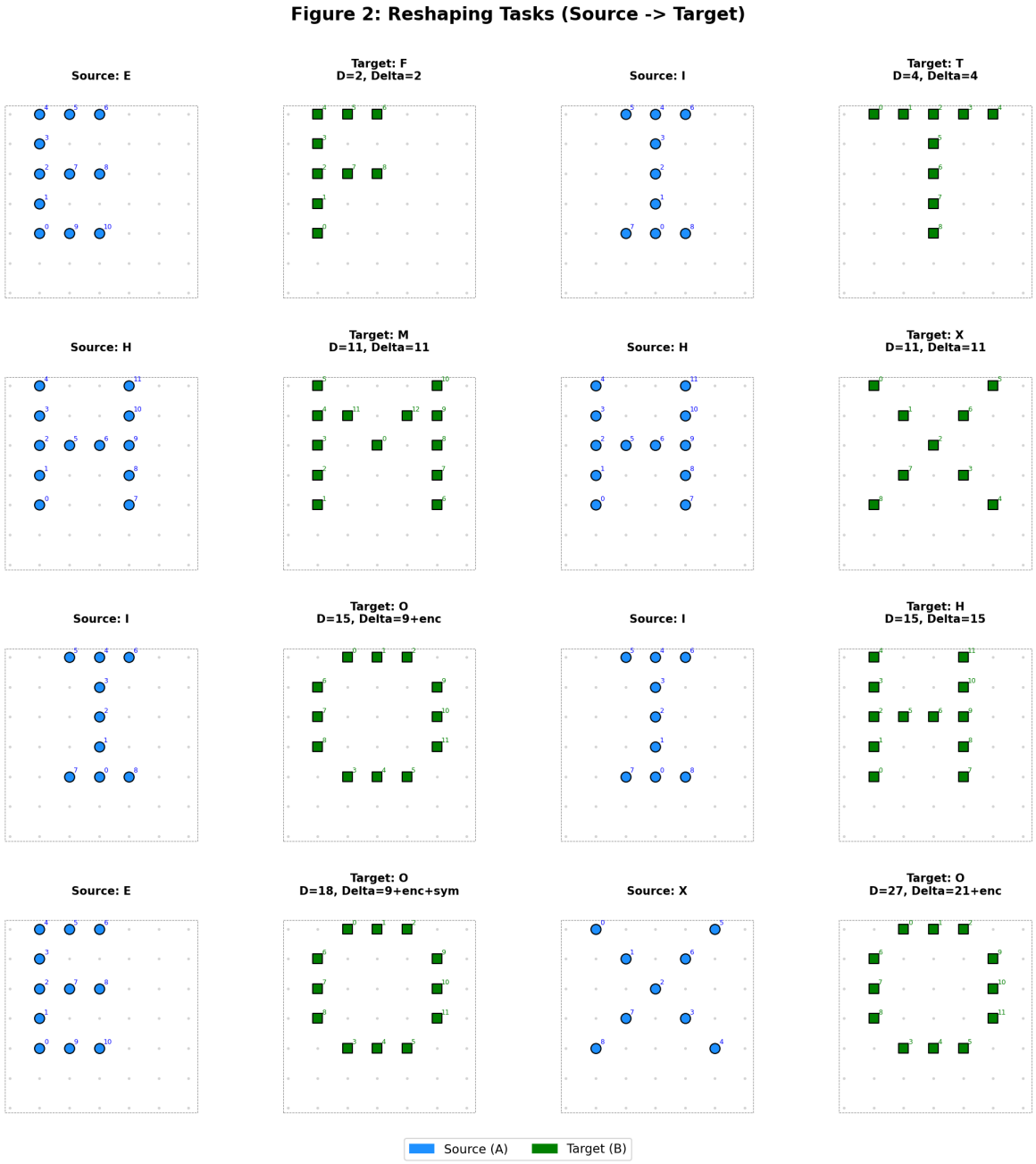

Figure 5: **Reshaping Tasks by Difficulty Level.** Each row shows a reshaping pair from the Easy, Medium, Hard, and Extreme categories. The composite difficulty D increases from left to right, incorporating both edit distance Δ and topological penalties for enclosure and symmetry transitions.

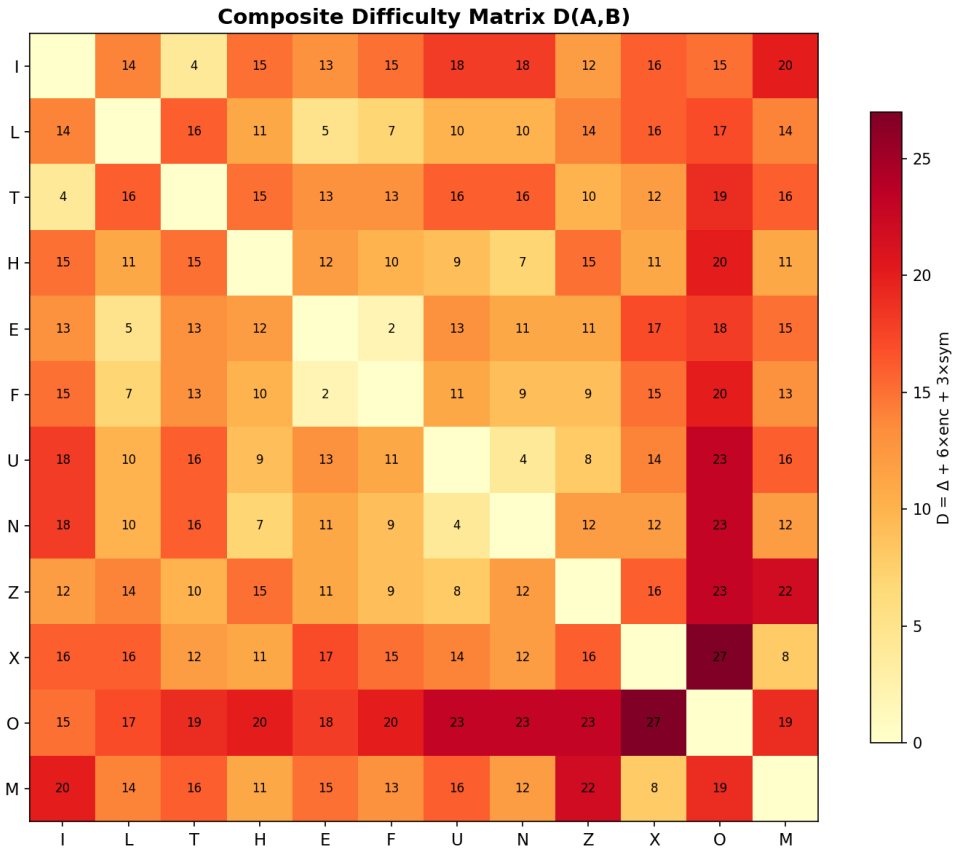

Figure 6: **Difficulty Matrix Heatmap.** Composite difficulty D for all 66 reshaping pairs. Darker colors indicate greater difficulty. The enclosure shape O shows consistently great difficulty when paired with non-enclosure shapes due to the $+6$ topology penalty.

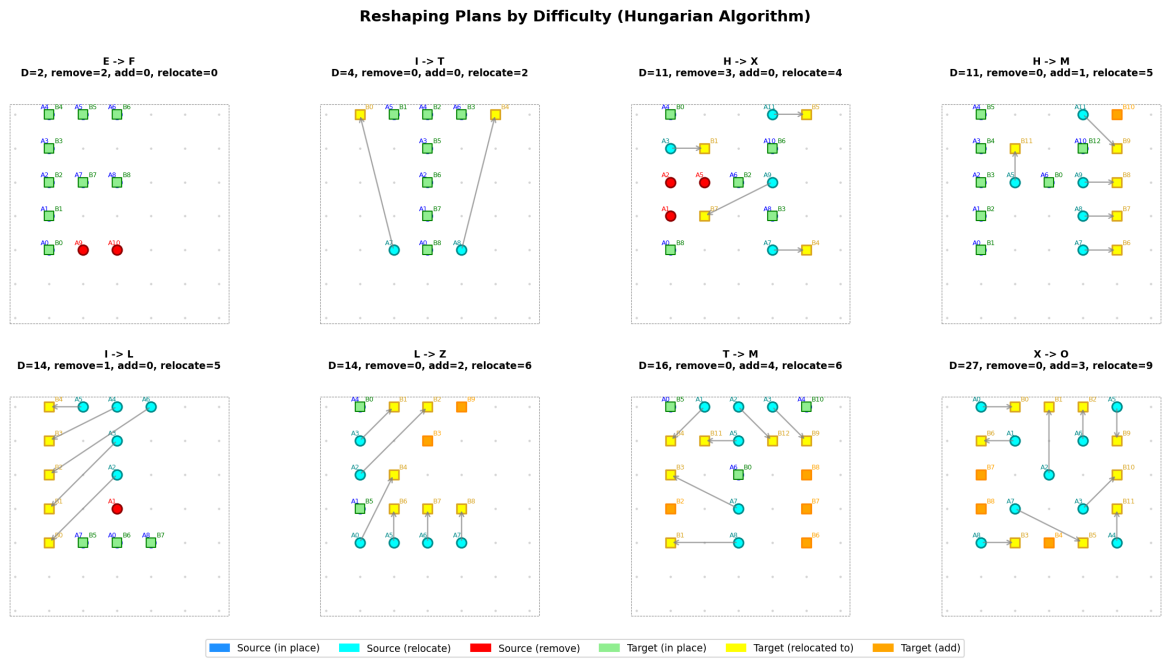

Figure 7: **Optimal Reshaping Plans via Hungarian Algorithm.** Each subplot shows the transformation plan for a reshaping pair. Colors indicate: blue/cyan = source positions (in place/relocate), green/yellow = target positions (in place/relocated to), red = cubes to remove, orange = positions to add. The Hungarian Algorithm minimizes total movement distance while computing optimal cube-to-position assignments.

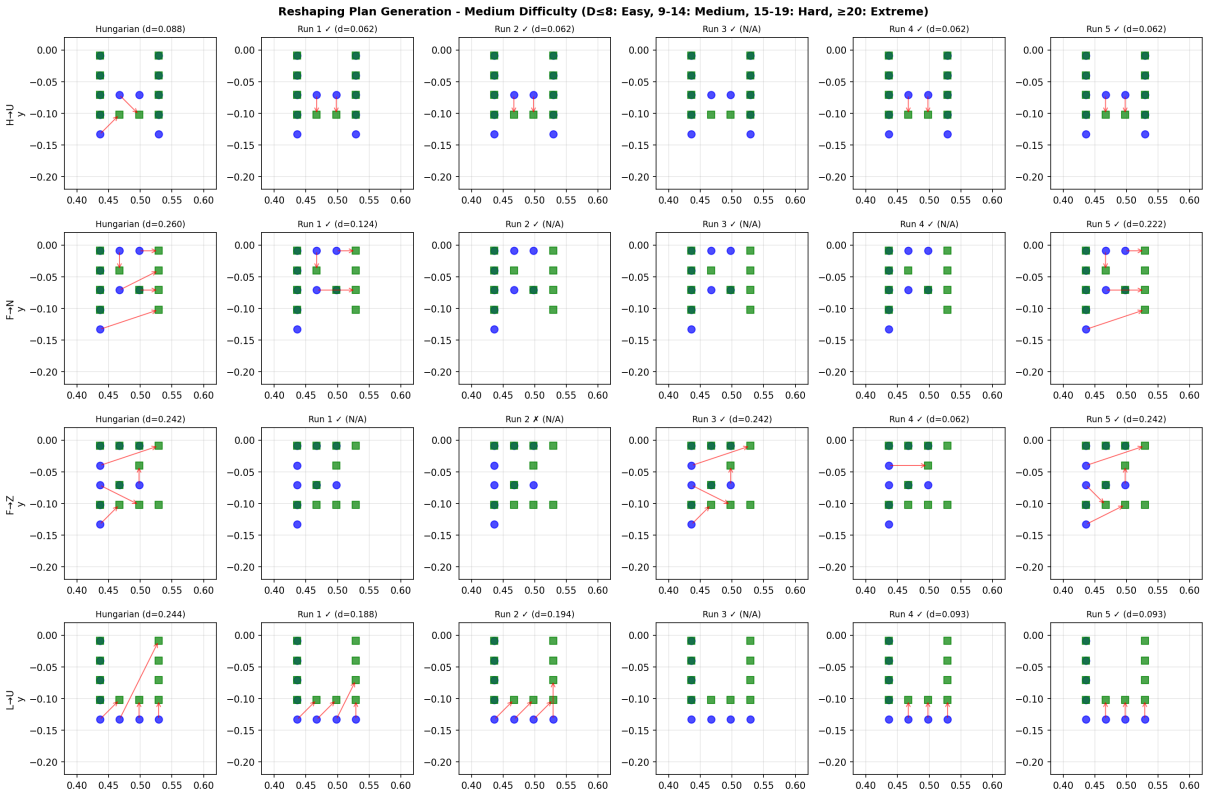

Figure 8: Reshaping tasks at Medium difficulty ($9 \leq D \leq 14$): larger Δ, may include symmetry flips, no enclosure transitions. Representative subset of pairs at this difficulty bucket; Easy pairs are shown earlier in Figure 5.

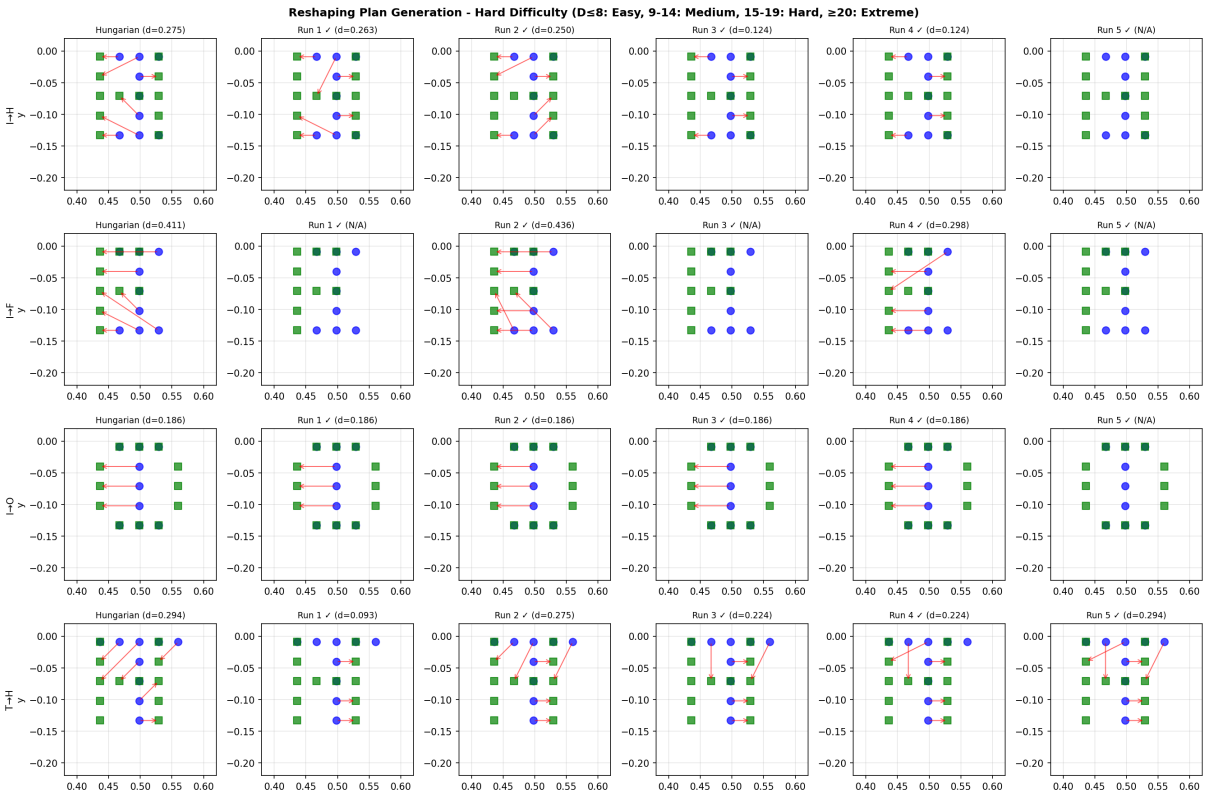

Figure 9: Reshaping tasks at Hard difficulty ($15 \leq D \leq 19$): may involve enclosure transitions or large Δ combined with symmetry changes.

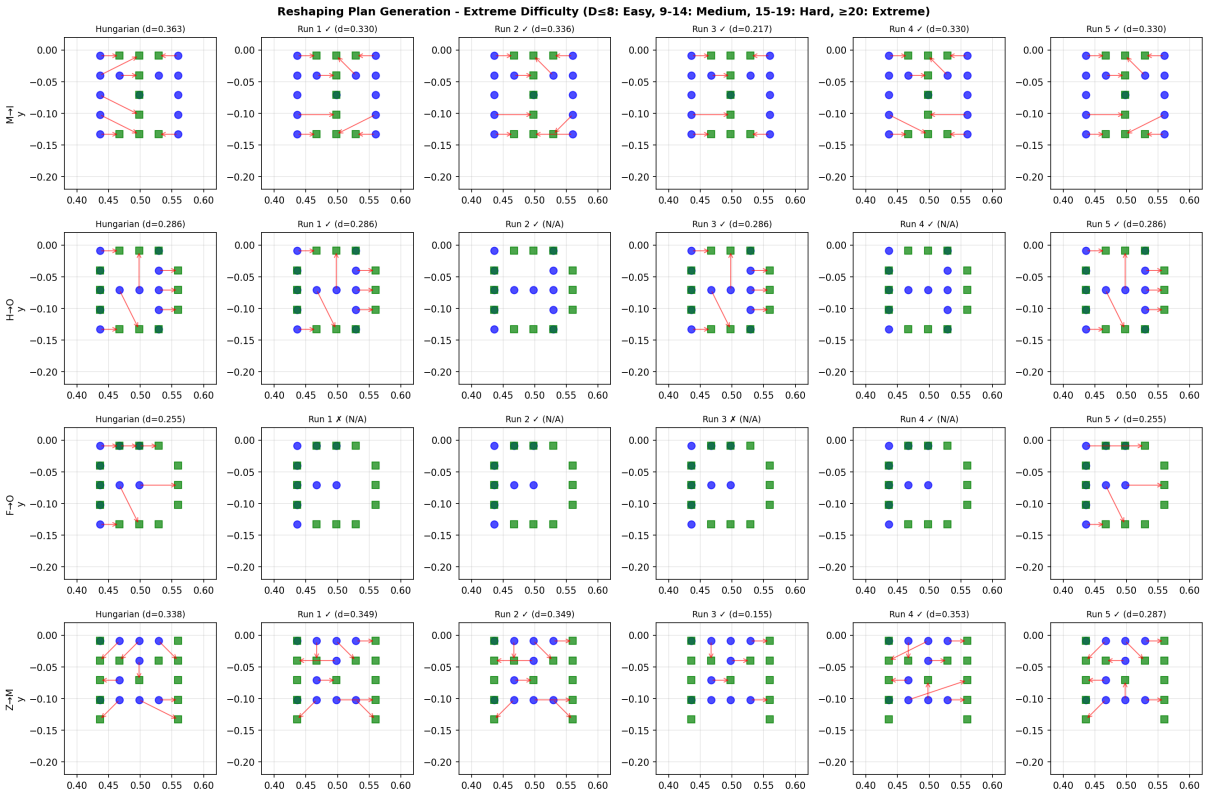

Figure 10: Reshaping tasks at Extreme difficulty ($D \geq 20$): combine large Δ with multiple topology changes (enclosure and/or symmetry).