WORLDENGINE: REVERSE ENGINEERING VISUAL WORLDS WITH EXECUTABLE CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Adapting AI models to novel visual domains typically requires humans to hand-engineer simulators, training data, and evaluators, whereas learned neural world models remain difficult to inspect, verify, or control. We introduce WORLDENGINE, a framework in which a coding agent turns a minimal task description and sparse visual examples into an executable *world hypothesis*: an engine that generates task instances, renders solution trajectories, and validates candidate solutions, with configurable difficulty and appearance. In Part I, evaluations across four visual reasoning and planning domains expose an *executability-fidelity gap*: engines that run without error still encode wrong rules, leak answers, or collapse controls into fixed templates. Our structured construction workflow improves rule fidelity and validity, and explicit diversity objectives activate controllable variation. In Part II, we use a synthesized engine to adapt a video-based visual planner in two regimes. Used *open-loop*, the engine supplies verified trajectories for fine-tuning, raising the solving rate on an independent simulator's puzzles from 2% to 24%. Used *closed-loop*, its validator feeds per-difficulty success back into sampling from a fixed pool, improving mean hard-level validation solving rate by up to 12.0 points over a static mix. Together, these results establish executable world reverse engineering as an inspectable, controllable engine for adapting visual planners.

1 INTRODUCTION

Adapting an AI system to a novel visual domain typically requires human engineers to supply the task rules, a simulator, training data, and a solution evaluator (OpenAI et al., 2024). This manual pipeline becomes a bottleneck when domain mechanics are only implicitly demonstrated through a few visual examples. The challenge is particularly acute in *visual planning* (Xu et al., 2025; Li et al., 2026a), where a conditional visual generative model plans directly in pixel space by synthesizing a sequence of frames that transforms an initial observation into a goal state (Du et al., 2023), e.g., sliding blocking vehicles to clear an exit in RUSHHOUR (Zeller et al., 2026). Turning a visual generator into a planner requires *rule-consistent trajectory demonstrations* for supervision and a way to verify candidate solutions: both are exactly what a hand-built simulator provides, and what a new domain lacks.

Can an agent instead reverse-engineer its own playable, verifiable training environment from sparse visual evidence? Neural world models predict future frames in latent space (Ha & Schmidhuber, 2018; Hafner et al., 2023), but the dynamics they learn are encoded in distributed parameters that are hard to inspect, verify, or control. Hand-crafted simulators provide exact transitions, validators, and controls, yet shift the specification burden back to humans. We explore a middle path: WORLDENGINE asks a coding agent to translate a minimal description and a few visual observations into an executable *world hypothesis*, a program that states the inferred rules explicitly and can therefore generate new task instances with reference solutions, render them as images or videos, and verify candidate solutions, with adjustable difficulty and appearance. We call such a program a *world engine*.

A handful of visual examples can be explained by many different underlying rules, so reverse-engineering a world engine requires reasoning about the semantics behind the pixels rather than merely reproducing them. Auditing agent-built engines, we find an executability-fidelity gap: an agent can easily write a program that executes without error and renders visually convincing boards, yet silently fails as a training environment for visual planning. We observe three failure modes: (1) *rule hallucination*, where inferred rules violate motion or collision constraints; (2) *answer leakage*,

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

where outputs reveal solutions through metadata or visual shortcuts; and (3) *collapsed controls*, where difficulty and style parameters merely select among a few fixed templates. WORLDENGINE addresses these failures with a domain-agnostic *construction skill workflow*: the agent must first write its hypothesis down explicitly, implement it together with executable tests, and submit it to an independent critic whose feedback routes it back to revising the rules or repairing the code; an explicit diversity objective targets collapsed controls.

In this paper, we study both how to reverse-engineer controllable world engines (Part I) and how to exploit them for downstream visual planning (Part II). In Part I, across four visual puzzle domains (HER-TREES, CLOTH NESTING, TANGRAM (Li et al., 2026a), and RUSHHOUR (Zeller et al., 2026)), we evaluate engine executability, rule fidelity, information validity, controllability, and visual quality. Ablations on HER-TREES show that the construction skill raises rubric-scored engine quality, and that without an explicit diversity objective the control parameters collapse no matter how many examples the agent sees. In Part II, we use the engine reverse-engineered for RUSHHOUR to train a 5B video generation model, Wan 2.2 TI2V-5B (Wan et al., 2025), as a visual planner in two modes: (1) *open-loop* supervised fine-tuning (WE-SFT) on engine-generated trajectories, which raises the solving rate on out-of-distribution puzzles from an independent simulator from 2% to 24%; and (2) *closed-loop* curriculum allocation, where, instead of sampling the difficulty levels at a constant ratio, the engine's validator measures per-level success during training and an EMA or Kalman controller re-weights the mix accordingly, improving the mean solving rate on hard-level validation puzzles by up to 12.0 percentage points. Our main contributions are threefold:

- **Executable World Reverse Engineering:** We propose WORLDENGINE, a framework that turns sparse visual examples into executable, configurable world engines through rule inference, code generation, and critic-guided revision.
- **Auditing the Executability-Fidelity Gap:** Four-domain audits characterize rule errors, answer leakage, and collapsed controls. Ablations show that the construction skill and explicit diversity objectives each improve engine quality and controllability.
- **Open- and Closed-Loop Visual Planning Adaptation:** On RUSHHOUR, we demonstrate cross-engine OOD gains via open-loop SFT and improved average hard-level controller-validation performance via closed-loop curriculum allocation.

2 WORLDENGINE: SYNTHESIZING AND EXPLOITING WORLD HYPOTHESES

2.1 PROBLEM FORMULATION: PROGRAMMATIC WORLD HYPOTHESES

During world reverse engineering, a coding agent is given a code execution environment and only three task-related inputs: a brief natural-language task description d, a small set of visual observations $\mathcal{O}_{\text{obs}}$ (e.g., 1-6 visual examples either images or videos), and a domain-agnostic construction skill Π. The visual examples ground task-specific physical and geometric constraints left implicit in the description. No other task-specific evidence is provided; in particular, none of the evaluation data used in Sections 3 and 4 is visible to the agent.

WORLDENGINE tasks the agent with synthesizing an executable *world hypothesis* $\hat{W}$ in code (Tang et al., 2024; Dainese et al., 2024). Everything else, including the state semantics and constraints, a generator with its solver, a renderer, a validator, and the control axes, must be inferred from d and $\mathcal{O}_{\text{obs}}$ and implemented by the agent. Programmatically, $\hat{W}$ exposes three interfaces, with generation parameterized by a vector of configuration controls $\theta \in \Theta$:

$$\texttt{Generate}(\theta) \to (s_0, \tau^\star), \qquad \texttt{Render}(\tau) \to V_{\text{vid}}, \qquad \texttt{Validate}(s_0, \hat{y}) \to \{0, 1\}. \tag{1}$$

Given controls θ, `Generate` samples an initial state s_0 and a reference trajectory $\tau^\star$. `Render` maps a trajectory to a video, or a single state to an image ($I_0 = \texttt{Render}(s_0)$). `Validate` checks a candidate solution $\hat{y}$ (a trajectory $\hat{\tau}$ for planning tasks) with a task-specific evaluator.

Because finite observations $\mathcal{O}_{\text{obs}}$ do not uniquely identify the true environment, we evaluate a synthesized engine along four progressive criteria: (1) Executability (whether the three interfaces run reliably and conform to the contract above); (2) Rule Fidelity (whether inferred transitions and constraints match the true task mechanics); (3) Information Validity (whether outputs are free of answer-leaking metadata or trivial visual shortcuts); and (4) Controllability (whether varying θ induces structural and stylistic variation across generated instances).

2.2 SYNTHESIZING AND FALSIFYING WORLD ENGINES

When task specifications are incomplete, coding agents frequently synthesize self-consistent programs that render plausible visuals yet violate a rule, leak an answer, or ignore a control, which should be revised. Our construction skill Π addresses this through three core mechanisms (Figure 1, Left).

Explicit Hypothesis Articulation. Before implementing rendering or generation logic, the skill requires the agent to formulate an *explicit hypothesis*: the inferred state variables, preconditions of legal transitions, invariants, and goal predicates. The agent then translates these rules into the `Generate`, `Render`, and `Validate` contract, with executable unit tests (Chen et al., 2023) and a few synthesized demonstration trajectories. Stating the hypothesis before writing code separates what the rules are from how they are implemented, allowing the critic to review each independently.

Automated Criticism with Dual-Path Routing. An independent critic (Madaan et al., 2023; Shinn et al., 2023; Zheng et al., 2023) audits source code and rendered artifacts across six review axes that operationalize the criteria above: task representation, logic alignment, complexity control, output diversity, format validity, and implementation quality. Rather than returning a single verdict, the critic diagnoses where a failure originates and routes the agent accordingly:

- Structural Failures $\rightarrow$ Rule Inference: If the hypothesis itself misstates physical constraints, visual identifiers, or information that should be hidden from the solver, the critic rejects it and sends the agent back to revise the hypothesis.
- Implementation Bugs $\rightarrow$ Code Repair: If the hypothesis is sound but the program raises runtime exceptions, violates the output format, or disagrees with its own evaluator, the critic sends the agent to repair the code.

The skill budgets three outer construction/review iterations and up to six rule-search iterations; recorded execution counts are reported in Appendix B.

Inducing Orthogonal Control Axes (θ). We prompt for separate controls over structural difficulty (e.g., obstacle density, solution depth) and visual appearance (e.g., palette, brush texture), in the spirit of procedural generation and domain randomization (Cobbe et al., 2020; Tobin et al., 2017). The diversity objective asks the agent to implement these controls and demonstrate their effects in generated cases. Without it, controls tend to collapse into a few fixed templates (Section 3.3).

2.3 EXPLOITING $\hat{W}(\theta)$ FOR DOWNSTREAM VISUAL PLANNING

Once an engine $\hat{W}$ passes falsification, its three programmatic interfaces are used to adapt a downstream visual planner (Figure 1, Right). In visual planning, a conditional visual generative model π_ϕ (e.g., video generation model (Wan et al., 2025)) receives a rendered initial board image $I_0 = \texttt{Render}(s_0)$ and a fixed task prompt p, and generates a visual rollout $\hat{V}_{\text{vid}} \sim \pi_\phi(\cdot \mid I_0, p)$. A state parser extracts the implied state trajectory $\hat{\tau}$ from $\hat{V}_{\text{vid}}$ (Appendix F), so that $\texttt{Validate}(s_0, \hat{\tau})$ determines planning success automatically. We investigate two exploitation modes that differ in whether the planner's performance feeds back into how the engine is used.

Open-Loop Exploitation. In the open-loop mode, the engine is queried once to produce training data and receives no feedback from the visual planner being trained (We-SFT). Sampling controls θ, calling $(s_0, \tau^\star) = \texttt{Generate}(\theta)$, keeping only instances with $\texttt{Validate}(s_0, \tau^\star) = 1$, and rendering $V^\star_{\text{vid}} = \texttt{Render}(\tau^\star)$ yields synthetic visual planning demonstrations. Supervised fine-tuning of π_ϕ on the pairs $(I_0, V^\star_{\text{vid}})$ tests whether mechanics reverse-engineered from sparse examples transfer out of distribution. At test time, the planner is evaluated on puzzles generated by an independent, human-written simulator that the visual planner has never seen (*cross-engine OOD*).

Closed-Loop Exploitation. Because the engine both controls problem complexity and automatically verifies candidate solutions via `Validate`, we can measure the planner's solving rate at each difficulty level during training with a held-out synthetic instances as a *controller-validation set* and adjust the training mix online. At periodic training intervals t, we evaluate π_ϕ on this set with `Validate`, obtaining per-level success rates $y_{t,k}$. A curriculum controller, e.g., an Exponential Moving Average (EMA) or a Kalman filter (Kalman, 1960), maintains an estimate $\hat{x}_{t,k}$ of the planner's mastery of each level and re-allocates the next stage's exposure budget (n_2, n_3) toward the level whose mastery lags (Appendix E.1). This follows automatic curriculum learning (Graves et al., 2017;

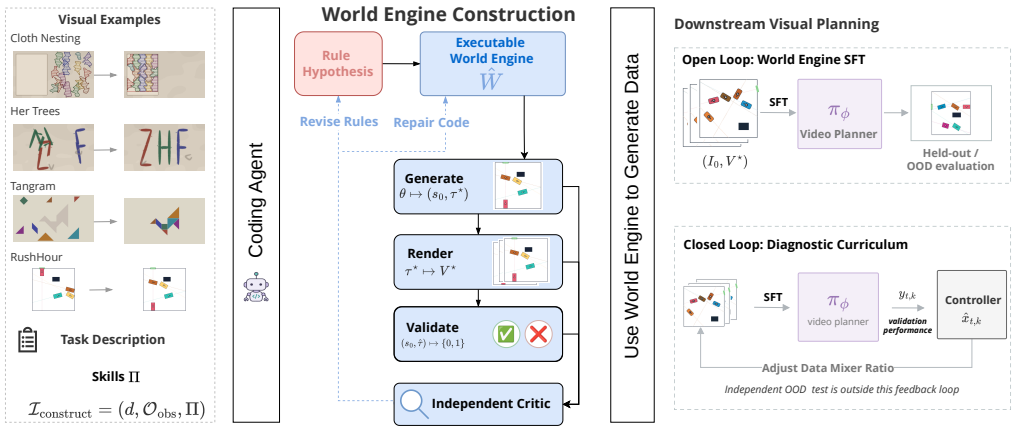

Figure 1: **WORLDENGINE pipeline.** *Left:* from task demonstrations, a coding agent writes an explicit world hypothesis, implements its code contracts, and routes failures back for revision. *Right:* the resulting engine supplies verified trajectories for open- and closed-loop SFT for visual planner.

Matiisen et al., 2017; Portelas et al., 2020), except that the difficulty labels, the training instances, and the success signal are all supplied by the reverse-engineered engine.

3 REVERSE ENGINEERING VISUAL WORLDS

We first evaluate whether coding agents can synthesize runnable, visually realistic, and controllable world engines from sparse visual observations across four domains, and ablate the design choices that govern synthesis quality.

3.1 EXPERIMENTAL SETUP

Task Domains. We evaluate world reverse engineering across four visual reasoning and planning puzzles with complementary linguistic and visual specifications:

- **HER-TREES:** Group motion reasoning where objects sharing the same visible color must translate and rotate together as a rigid group without colliding or splitting.
- **CLOTH NESTING:** 2D irregular geometric packing where polygonal garment pieces must be placed inside a bounding container without overlap.
- **TANGRAM** (Li et al., 2026a): Geometric shape decomposition requiring rigid rotation and translation of primitives to tile a target silhouette.
- **RUSHHOUR** (Zeller et al., 2026): Sequential track-constrained planning where vehicles slide along fixed horizontal or vertical tracks without collision to clear an exit path for the target car.

Coding Agents and Input/Output Contract. We evaluate WORLDENGINE across three coding-agent harnesses on all four domains: Claude Code (Opus 4.8), Codex CLI (GPT-5.5, high), and OpenCode (GPT-5.5, default); CLI versions and runtime details are reported in Appendix A.3. For each domain, the agent receives only a brief natural-language description d, 1-6 visual examples $\mathcal{O}_{\text{obs}}$, and the construction skill Π (iteration budgets in Appendix B). Each run must output a self-contained Python package implementing `Generate`(θ), `Render`$(\cdot)$, and task-specific `Validate`, accompanied by executable unit tests and at least three rendered trajectory videos.

Evaluation Metrics. We assess synthesized engines along three complementary protocols (detailed rubrics in Appendix D): (1) an external agent-based 4-axis rubric scoring *Rule Fidelity* (F), *Executability* (E), *Controllability* (C), and *Information Validity* (I) on a 1-5 scale (with each OpenCode build independently audited by GPT-5.5-high); (2) a source-blinded VLM preference test, where Gemini-3.5-Flash judges 30 matched initial-state image pairs against human-crafted reference simulators; and (3) automated control audits that count active configuration axes (parameters taking multiple realized values) and distinct visual style families in generated metadata.

3.2 MULTI-TASK ENGINE QUALITY

Programmatic quality and the executability-fidelity gap. Across domains and coding agents, high code executability does not guarantee faithful task mechanics or valid information hiding, confirming a pronounced *executability-fidelity gap* (Table 1). Across the 12 evaluated builds, agents reliably produce runnable programs, yielding the highest mean score on Executability (4.58/5.0), whereas Rule Fidelity (3.67/5.0) and Information Validity (3.50/5.0) lag substantially behind. This gap is especially clear in the OpenCode builds: all four domains pass standard software execution checks, yet only HER-TREES and RUSHHOUR pass both internal criticism and external review. In the remaining builds, CLOTH NESTING leaks ground-truth placement hints through answer-bearing geometry, while TANGRAM produces disconnected, label-ambiguous assemblies (Figure 2). Therefore, passing unit execution checks alone is insufficient to establish a valid training environment.

Critic-guided repair. Iterative critic-guided falsification bridges this gap by catching and repairing semantic-visual inconsistencies that standard unit tests miss. For example, in the accepted Claude TANGRAM trace (Appendix C), the initial implementation passes its unit test suite, but Critic 3 detects that an interior polygon hole vanishes during rendering, causing the engine's own reference assembly to achieve an IoU of only 0.9412 against the 0.98 acceptance threshold. Routed back for code repair, the builder revises generation to filter out holed topologies and adds regression tests; Critic 4 subsequently verifies a gold IoU of 1.0 across 24 test cases and 22 passing tests before accepting the build. Conversely, when an agent fails to resolve structural defects within the iteration budget, such as the disconnected TANGRAM assembly in Figure 2, the critic properly rejects the engine rather than admitting a flawed simulator.

Visual realism against reference engines. Beyond programmatic validity, synthesized engines render visual observations that match or surpass human-crafted reference simulators in source-blinded perceptual evaluations. Across 30 matched image pairs judged by Gemini-3.5-Flash, WORLDENGINE is preferred in 60.0% (18/30) of comparisons overall. Performance is strongest on CLOTH NESTING ($13/15$ wins), where procedural rendering introduces richer fabric textures and layout clarity, while remaining competitive on the more stylistically constrained HER-TREES domain ($5/15$ wins; representative rollout frames and per-task breakdowns are shown in Appendix Figure 5).

3.3 ABLATION EXPERIMENTS

To isolate the factors driving engine quality and controllability, we conduct controlled ablations on HER-TREES using Codex (GPT-5.5, $n = 3$ independent runs per condition, 24 runs total; Table 2) along three axes: (1) *Workflow structure* (Π vs. No-Skill), removing the construction skill at 3-shot to test the impact of explicit hypothesis articulation and critic routing; (2) *Prompting objective* (Neutral vs. Diversity-Explicit), varying whether orthogonal difficulty and style controls are explicitly requested; and (3) *Visual evidence regime*, scaling positive demonstrations (1, 3, or 6 examples) and introducing conflicting negative examples (6 positive plus 3 negative cases from an incompatible rule) to test how observation volume and noise affect rule recovery.

Effect of the Construction Skill (Π). Removing the construction skill (Π) causes severe degradation in both visual rendering and information hiding, demonstrating that unstructured agent coding cannot reliably close the executability-fidelity gap. Under the 3-shot neutral setting, equipping the agent with Π raises the mean rubric score from 3.00 (*No-Skill*) to 3.92 (Table 2). Without the structured hypothesis-test-critic loop, generated engines frequently output blank or illegible boards and leak ground-truth answers in case metadata.

Effect of Diversity Objectives. Explicit diversity objectives are essential to prevent control collapse and unlock multi-axis procedural variation. Across all example budgets (1, 3, and 6-shot), diversity-explicit prompting consistently activates 6–7 independent control axes, spanning difficulty, brush, palette, scatter, and rotation, and yields 4 distinct visual style families (Table 2, Figure 3). At 3-shot, adding the diversity objective also improves overall rubric quality from 3.92 to 4.75 by encouraging cleaner parameterization between task structure and rendering style.

Effect of Example Budget and Negative Distractors. Scaling visual examples alone fails to induce controllability under neutral prompting, while conflicting negative distractors can trigger subtle rule hallucinations on complex instances. First, under neutral prompting, increasing positive examples from 1 to 3 and 6 leaves the mean number of active control axes near zero (1.00, 0.00, and

Table 1: **Multi-domain world engine quality across four tasks and three coding-agent harnesses.** External agent reviewers score each synthesized build from 1 to 5 on evaluation protocols, details in Appendix D. While agents consistently achieve high *Executability*, substantially lower scores on *Rule Fidelity* and *Information Validity* expose a persistent *executability-fidelity gap*.

Task Domain	Coding Agent	Fidelity	Executability	Controllability	Information Validity	Average
HER-TREES	Claude Code (Opus)	4	5	5	3	4.25
	Codex (GPT-5.5)	3	5	4	4	4.00
	OpenCode (GPT-5.5)	4	5	4	4	4.25
CLOTH NESTING	Claude Code (Opus)	5	4	4	4	4.25
	Codex (GPT-5.5)	3	5	4	3	3.75
	OpenCode (GPT-5.5)	3	3	4	1	2.75
TANGRAM	Claude Code (Opus)	5	4	4	5	4.50
	Codex (GPT-5.5)	3	5	3	3	3.50
	OpenCode (GPT-5.5)	2	4	3	1	2.50
RUSHHOUR	Claude Code (Opus)	4	5	4	5	4.50
	Codex (GPT-5.5)	4	5	4	4	4.25
	OpenCode (GPT-5.5)	4	5	4	5	4.50
Mean across 12 evaluated builds		3.67	**4.58**	3.92	3.50	3.92

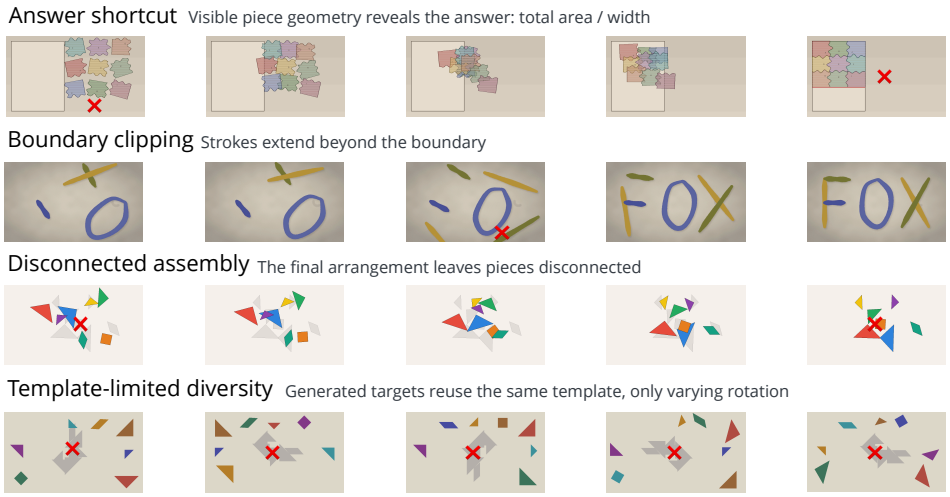

Figure 2: **Representative artifacts and failure modes across synthesized world engines.** Unverified builds exhibit (1) *answer shortcuts* with leakage (CLOTH NESTING), (2) *rendering/boundary clipping* (HER-TREES), (3) *structural rule violations* such as disconnected primitive assemblies (TANGRAM, rejected by our internal critic), and (4) *collapsed diversity* under template-bound generation.

0.33), showing that agents default to fixed templates regardless of example count unless explicitly prompted for diversity. Second, adding 3 labeled negative distractors to the 6-shot diversity setting preserves high executability (4.67) and control diversity (7.00 axes), but slightly depresses rule fidelity on hard instances—for example, producing boards where eight independently moving groups share only six visible colors, violating the same-color rigid-motion invariant (Figure 3, right; full per-condition distributions and image-level diagnostics are reported in Appendix D.4 and Appendix Figure 6).

4 EXPLOITING WORLD ENGINES FOR VISUAL PLANNING

Having evaluated engine synthesis in Part I, we now test whether a reverse-engineered world engine can effectively adapt a downstream visual planner. While HER-TREES offers rich stylistic variation for auditing engine construction, RUSHHOUR demands multi-step sequential reasoning with graded

Table 2: **Ablation study on HER-TREES across construction skill (Π), prompting objectives, and visual example budgets** (Codex GPT-5.5; mean over $n = 3$ runs per condition, 24 runs total). Ex. denotes the number of positive (and labeled negative, 6+3) visual examples; F/E/C/I denote 1–5 rubric scores for the evaluation protocol. Axes counts configuration parameters taking multiple realized values; and Styles counts distinct brush/style families in generated metadata.

		Engine Rubric Quality (1–5)					Realized Controls	
Prompting	Ex.	F	E	C	I	Avg (↑)	Axes (↑)	Styles (↑)
Baseline without Construction Skill (Π removed)								
Neutral, no skill	3	2.67	3.67	2.67	3.00	3.00±1.30	0.33	0.00
With Construction Skill (Π) — Neutral Prompting								
Neutral	1	4.33	5.00	3.67	4.33	4.33±.38	1.00	0.00
Neutral	3	3.33	4.67	4.00	3.67	3.92±.38	0.00	0.00
Neutral	6	4.00	5.00	3.33	3.67	4.00±.43	0.33	0.00
With Construction Skill (Π) — Diversity-Explicit Prompting								
Diversity-explicit	1	4.33	5.00	4.33	**5.00**	4.67±.29	**7.00**	**4.00**
Diversity-explicit	3	4.33	5.00	4.67	**5.00**	**4.75±.25**	**6.00**	**4.00**
Diversity-explicit	6	3.67	4.67	**5.00**	4.00	4.33±.52	**7.00**	**4.00**
Diversity + negative	6+3	4.00	4.33	**5.00**	4.67	4.50±.25	4.33	3.00



Figure 3: Visual comparison of HER-TREES engines under neutral versus diversity-explicit prompt.

solution depths, making it an ideal testbed for visual planning (Zeller et al., 2026; Geirhos et al., 2026). Using the synthesized RUSHHOUR engine and Wan2.2 TI2V-5B (Wan et al., 2025) as our base video generator, we evaluate both open-loop and closed-loop exploitation. Throughout this section, we measure planning performance by *solving rate*: the fraction of test puzzles for which the discrete move sequence parsed from a single generated video rollout legally reaches the goal state under the task validator (parsing and inference details in Appendix F). To rigorously test generalization beyond the synthesized engine's own rendering and layout distribution, we evaluate trained planners on unseen *cross-engine OOD* puzzles generated by an independent, human-crafted reference simulator.

4.1 OPEN-LOOP EXPLOITATION: TRAJECTORY FINE-TUNING

In the open-loop training regime (WE-SFT), we query the synthesized RUSHHOUR engine to generate initial states and reference trajectories via `Generate`(θ) across medium (Level-2) and hard (Level-3) difficulty settings, filter verified solutions with `Validate`, and render demonstration videos via `Render` to fine-tune Wan2.2 TI2V-5B. Table 3 evaluates the base model and two WE-SFT checkpoints (Epochs 10 and 16) on 100 held-out cross-engine OOD puzzles from the reference simulator (50 Level-2 and 50 Level-3), alongside off-the-shelf video generation baselines.

Synthetic trajectories transfer zero-shot to independent test cases. Fine-tuning solely on trajectories from the reverse-engineered engine yields substantial cross-engine OOD transfer, raising the overall solving rate on the human-crafted reference simulator from 2% (base model) to 18% at Epoch 10 and 24% at Epoch 16—a 12× improvement (Table 3). Because the planner never observes specific frames or data points from the reference simulator during training, this gain confirms that the synthesized engine captures transferable rules and state transitions, and demonstrates the utility of engine-generated supervision for cross-engine visual-planning adaptation.

Asymmetric learning dynamics across difficulty levels motivate adaptive curricula. Breaking performance down by difficulty reveals that medium puzzles saturate early while hard puzzles remain

Table 3: **Open-loop visual planning on RUSHHOUR (L2/L3).** Top rows evaluate Wan2.2 and WE-SFT on 100 unseen puzzles (50 per level) from an independent reference simulator; bottom rows show published baselines under their original protocols. WE-SFT boosts cross-engine solving rate from 2% to 24% (18% on L3).

Method	L2 solving rate	L3 solving rate	Overall
Wan2.2 Base (zero-shot)	1/50 (2%)	1/50 (2%)	2/100 (2%)
WE-SFT (Epoch 10)	**15/50 (30%)**	3/50 (6%)	18/100 (18%)
WE-SFT (Epoch 16)	**15/50 (30%)**	**9/50 (18%)**	**24/100 (24%)**
Wan2.6 (MentisOculi) (Zeller et al., 2026)	3/30 (10%)	2/30 (6.7%)	—
VIPE Veo3.1 (Geirhos et al., 2026)	≈48%	≈8%	—
VIPE Wan2.2 (Geirhos et al., 2026)	≈5%	1–5%	—

† VIPE rates are inferred for the best of 20 variants from 10 puzzles per level with 10 videos each.

bottlenecked under static training. Specifically, the Level-2 solving rate reaches 30% (15/50) by Epoch 10 and plateaus at 30% at Epoch 16, whereas the Level-3 solving rate climbs more slowly from 6% (3/50) at Epoch 10 to 18% (9/50) at Epoch 16, accounting for all net gains between the two checkpoints. This divergence indicates that uniform sampling wastes capacity on already-mastered easier instances late in training, motivating closed-loop difficulty allocation.

4.2 CLOSED-LOOP EXPLOITATION: VALIDATOR-GUIDED CURRICULUM

To dynamically allocate training effort, we close the loop between the engine's difficulty control ($\theta \in \{\text{L2}, \text{L3}\}$) and its automated validator (`Validate`). Operating over a fixed engine-generated pool of 1,000 L2 ($\mathcal{D}_2$) and 1,000 L3 ($\mathcal{D}_3$) instances with a constant budget of 2,000 sampled instances per epoch, we compare a static *Fixed* uniform sampling baseline against two closed-loop curriculum controllers: *EMA* and *Kalman* (control equations in Appendix E.1). Every two epochs, the controllers evaluate π_ϕ via `Validate` on a held-out engine-generated validation set and reallocate the next epoch's per-level sampling budget (n_2, n_3) toward the lagging difficulty level (Table 4 and Figure 4).

Validator-guided reallocation accelerates hard-level mastery without cross-engine overfitting. Closing the loop via `Validate` automatically concentrates training budget on harder puzzles, boosting L3 mastery while preserving generalization to the external simulator (Table 4). Once L2 competence outpaces L3, both EMA and Kalman progressively shift the per-epoch sampling budget (n_3) toward L3 instances (Figure 4c). Across Epochs 4–20, this reallocation raises the mean L3 validation solving rate from 60.9% (Fixed 1:1) to 71.8% with EMA (+10.9 points) and 72.9% with Kalman (+12.0 points; Figure 4a). On the 50 external reference-simulator puzzles (Figure 4b), focusing on harder synthetic instances avoids domain overfitting and modestly improves mean cross-engine OOD solving rate from 29.8% to 32.0% (EMA) and 32.7% (Kalman), demonstrating closed-loop adaptation within a synthesized engine.

5 RELATED WORK

World Models. World models learn predictive latent dynamics for planning and control (Ha & Schmidhuber, 2018; Kim et al., 2020; Hafner et al., 2020; 2023; Bruce et al., 2024; Valevski et al., 2025). WORLDENGINE constructs an explicit executable hypothesis from sparse task evidence, exposing task rules, rendering, and validation to direct inspection and intervention.

Coding Agents. Coding agents combine editing, execution, and tests (Yao et al., 2023; Shinn et al., 2023; Chen et al., 2023) for software (Yang et al., 2024), visual inference (Ellis et al., 2020; Surís et al., 2023; Liang et al., 2023; Butt et al., 2024) and visual modelling (Wang et al., 2024a;b; Tang et al., 2024; Dainese et al., 2024; Lehrach et al., 2025; Chen et al., 2026; Fang et al., 2026; Rodionov, 2026; Li et al., 2026b; Zhang et al., 2026). Iterative self-feedback can also improve generated outputs (Madaan et al., 2023). WORLDENGINE couples task-specification inference with a shared generation, rendering, and validation contract. Critic findings guide revisions to the inferred rules and their executable implementation.

Table 4: **Closed-loop curriculum allocation on RUSHHOUR** averaged over Epochs 4-20 (Δ in percentage points). Validator-guided EMA and Kalman controllers raise mean L3 validation solving rate by $+10.9$ and $+12.0$ points over Fixed sampling while maintaining cross-engine OOD transfer.

Allocation Policy	L3 Validation Mean (%)	Δ vs. Fixed	Cross-Engine OOD Mean (%)	OOD Δ [95% CI]
Fixed (Uniform 1:1)	60.9	—	29.8	—
EMA Controller	71.8	+10.9	32.0	+2.2 [−4.4, +8.9]
Kalman Controller	**72.9**	**+12.0**	**32.7**	**+2.9** [−2.7, +8.7]

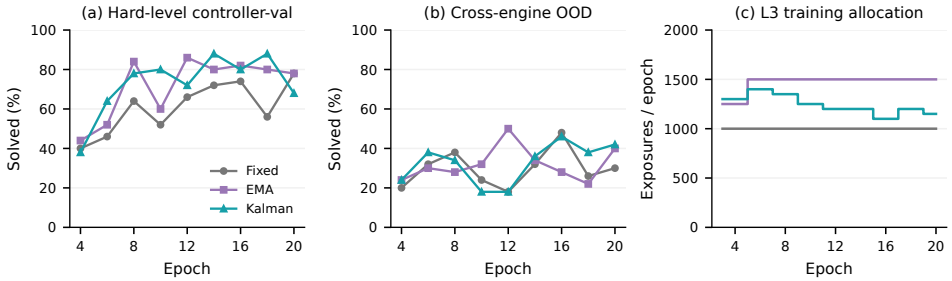

Figure 4: **Closed-loop curriculum dynamics (Epochs 4–20):** *(a)* L3 engine-validation solving rate, *(b)* external cross-engine OOD solving rate, and *(c)* per-epoch L3 sampling allocation (n_3). EMA and Kalman shift budget toward L3, boosting hard-level performance while preserving OOD transfer.

Executable World Models. WorldCoder builds Python world models from environment interactions (Tang et al., 2024). GIF-MCTS synthesizes and repairs code world models using textual descriptions and trajectories for model-based planning (Dainese et al., 2024). WORLDENGINE combines brief descriptions with sparse visual examples to construct a reusable training environment. We study generation, rendering, and validation jointly through construction audits and adaptation of a separate video planner, connecting executable world reconstruction to visual learning.

6 DISCUSSION

Synergy Between Programmatic Hypotheses and Pixel-Space Planners. The synthesized engine represents task states and legal transitions explicitly, computes reference solutions, and renders them as supervision. The video planner learns to produce solution trajectories from pixel observations, including puzzles from an external renderer. This connection turns the reconstructed world into a source of training data for visual-planner adaptation. On RUSHHOUR, engine-generated supervision raises the OOD solving rate of Wan2.2 from 2% to 24%.

From Inner-Loop Allocation to Outer-Loop World Revision. EMA and Kalman improve average hard-level controller-validation performance by 10.9 and 12.0 points within a fixed 2,000-instance pool. These results motivate extending allocation to adaptive instance generation: controllers could invoke `Generate`(θ) for new training cases and trigger revisions of $\hat{W}$ in response to downstream errors (Wang et al., 2019; Dennis et al., 2021; Parker-Holder et al., 2023).

7 CONCLUSION

WORLDENGINE turns sparse task evidence into executable world hypotheses with generation, rendering, and validation interfaces. Four-domain audits show high executability alongside rule errors, leakage, and template collapse. Ablation experiments on HER-TREES show that the construction skill improves rubric quality and explicit diversity objectives activate controllable variation. On RUSHHOUR, engine-generated trajectories raise cross-engine OOD solving rate from 2% to 24% through open-loop SFT. Closed-loop curriculum allocation improves average hard-level controller-validation performance by up to 12.0 percentage points. These results support executable world reverse engineering as an inspectable interface for domain adaptation.

AI USE STATEMENT

We used LLM to improve the grammar and readability of the manuscript. The authors reviewed and approved all edits. Generative AI was not used to develop the research questions or conclusions. The authors take responsibility for the final content of this paper.

REPRODUCIBILITY STATEMENT

The task inputs and construction prompts, evaluation protocols, and planner training settings are documented in Appendices A, D, and E, respectively. Together with the engine interfaces and metrics specified in the main text, these details provide a basis for reproducing the reported experiments. We will release our code and experimental materials upon publication to support independent reproduction.

REFERENCES

Jake Bruce, Michael Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, and et al. Genie: Generative Interactive Environments, February 2024.

Natasha Butt, Blazej Manczak, Auke Wiggers, Corrado Rainone, David W. Zhang, Michaël Defferrard, and Taco Cohen. CodeIt: Self-Improving Language Models with Prioritized Hindsight Replay, July 2024.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug, October 2023.

Yiwen Chen, Guosheng Lin, and Chi Zhang. Code World Model: Coding Agent as World Brain, August 2026.

Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning, July 2020.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating Code World Models with Large Language Models Guided by Monte Carlo Tree Search, October 2024.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design, February 2021.

Yilun Du, Mengjiao Yang, Bo Dai, Hanjun Dai, Ofir Nachum, Joshua B. Tenenbaum, Dale Schuurmans, and Pieter Abbeel. Learning Universal Policies via Text-Guided Video Generation, November 2023.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning, June 2020.

Zixun Fang, Yawen Shao, Kai Zhu, Jie Xiao, Shihan Chen, Yu Liu, Xueyang Fu, Yang Cao, Wei Zhai, and Zheng-Jun Zha. Code Plans, Diffusion Renders: Open-Ended Generative World Modeling, September 2026.

Robert Geirhos, Yuxuan Li, Thaddäus Wiedemer, Neha Kalibhat, Zi Wang, Mani Malek, Oyvind Tafjord, Kevin Swersky, Been Kim, and Priyank Jaini. Visual prompt engineering for video models, July 2026.

Alex Graves, Marc G. Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated Curriculum Learning for Neural Networks, April 2017.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination, March 2020.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35–45, March 1960. ISSN 0021-9223. doi: 10.1115/1.3662552.

Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. Learning to Simulate Dynamic Environments with GameGAN, May 2020.

Wolfgang Lehrach, Daniel Hennes, Miguel Lazaro-Gredilla, Xinghua Lou, Carter Wendelken, Zun Li, Antoine Dedieu, Jordi Grau-Moya, Marc Lanctot, Atil Iscen, and et al. Code World Models for General Game Playing, October 2025.

Chengzu Li, Zanyi Wang, Jiaang Li, Yi Xu, Han Zhou, Huanyu Zhang, Ruichuan An, Dengyang Jiang, Zhaochong An, Ivan Vulić, and et al. Thinking in Frames: How Visual Context and Test-Time Scaling Empower Video Reasoning, January 2026a.

Zhiqi Li, Yuxuan Liao, and Bo Zhu. Recursive Code World Models: Building Complex Worlds through Recursive Scene Programs, September 2026b.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as Policies: Language Model Programs for Embodied Control, May 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594, 2023.

Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher-Student Curriculum Learning, November 2017.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, and et al. GPT-4 Technical Report, March 2024.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving Curricula with Regret-Based Environment Design, September 2023.

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic Curriculum Learning For Deep RL: A Short Survey, May 2020.

Sergey Rodionov. Executable World Models for ARC-AGI-3 in the Era of Coding Agents. volume 16855, pp. 198–210. 2026. doi: 10.1007/978-3-032-33195-3_15.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning, October 2023.

Dídac Surís, Sachit Menon, and Carl Vondrick. ViperGPT: Visual Inference via Python Execution for Reasoning, March 2023.

Hao Tang, Darren Key, and Kevin Ellis. WorldCoder, a Model-Based LLM Agent: Building World Models by Writing Code and Interacting with the Environment, September 2024.

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World, March 2017.

Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. Diffusion Models Are Real-Time Game Engines, April 2025.

Team Wan, Ang Wang, Baole Ai, Bin Wen, Chaojie Mao, Chen-Wei Xie, Di Chen, Feiwu Yu, Haiming Zhao, Jianxiao Yang, and et al. Wan: Open and Advanced Large-Scale Video Generative Models, April 2025.

Lirui Wang, Yiyang Ling, Zhecheng Yuan, Mohit Shridhar, Chen Bao, Yuzhe Qin, Bailin Wang, Huazhe Xu, and Xiaolong Wang. GenSim: Generating Robotic Simulation Tasks via Large Language Models, January 2024a.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O. Stanley. Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions, February 2019.

Yufei Wang, Zhou Xian, Feng Chen, Tsun-Hsuan Wang, Yian Wang, Katerina Fragkiadaki, Zackory Erickson, David Held, and Chuang Gan. RoboGen: Towards Unleashing Infinite Data for Automated Robot Learning via Generative Simulation, June 2024b.

Yi Xu, Chengzu Li, Han Zhou, Xingchen Wan, Caiqi Zhang, Anna Korhonen, and Ivan Vulić. Visual Planning: Let's Think Only with Images, September 2025.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models, March 2023.

Jana Zeller, Thaddäus Wiedemer, Fanfei Li, Thomas Klein, Prasanna Mayilvahanan, Matthias Bethge, Felix Wichmann, Ryan Cotterell, and Wieland Brendel. MentisOculi: Revealing the Limits of Reasoning with Mental Imagery, June 2026.

Yi Zhang, Yunshuang Wang, Zeyu Zhang, and Hao Tang. Code2Worlds: Empowering Coding LLMs for 4D World Generation, February 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, December 2023. doi: 10.52202/075280-2020.

A CONSTRUCTION INPUTS AND PROMPTS

This appendix documents the supplied task specifications, construction instructions, recorded review interactions, evaluation protocols, and curriculum implementation. Prompt boxes distinguish complete text from excerpts. Markdown headings, lists, and code spans are typeset for readability without changing their wording. Host-specific absolute paths are replaced by the placeholders stated next to each prompt.

A.1 BUILDER INVOCATION AND INPUT LAYOUT

Each construction run receives a task description and supporting example files. The reference examples can include an input image, an assembled image, a labeled answer, and a solution video. Labels and reference trajectories are available to the builder for constructing the engine. The construction skill requires the engine to separate solver inputs from hidden answers and solution artifacts.

Prompt A.1. Builder invocation
Complete prompt from the saved Claude Tangram run.

Read ~/task/task_description.md. Use the build-world-engine skill to build a world engine from ~/task/examples and write it under ~/out/run-task.

The isolated non-Docker runner copies the task and example directory into the run's private home and installs the skill in the harness-specific skill directories. For example, the OpenCode run copies it under `.agents/skills/`. Its saved prompt appends an absolute-path contract because shell expansion of ~ may otherwise resolve outside that run. The following is the recorded suffix with paths normalized.

Prompt A.2. Isolated-run path contract
Complete recorded suffix. Angle-bracket paths replace absolute paths.

Runtime path contract (mandatory):

- Read the task only from: <TASK_DIR>
- Read the task description from: <TASK_DESCRIPTION_FILE>
- Write every run artifact only under: <RUN_DIR>
- The world engine path is: <WORLD_ENGINE_DIR>
- Do not use ~/task, ~/out, <HOST_HOME>/task, or <HOST_HOME>/out; those are shared paths outside this isolated run.
- Environment variables TASK_DIR, TASK_DESCRIPTION_FILE, RUN_DIR, and WORLD_ENGINE_DIR contain these exact absolute paths. Pass these absolute paths to the skill and all shell commands.

Appendix C documents recorded skill loading, example inspection, and critic interactions for two Tangram builds.

A.2 TASK-DESCRIPTION SNAPSHOTS

The following boxes reproduce the four task descriptions preserved with the OpenCode review artifacts (engines A–D, corresponding to Cloth Nesting, Her-Trees, Tangram, and RushHour). The Tangram text also matches the saved Claude Tangram description used in the repair case study. Each snapshot is associated with its source build. The wording, including inherited terminology in output-format comments, is preserved.

Prompt A.3. Cloth Nesting task description
Complete saved task description, engine A.

Task

Task Description

You are a tailor cutting pattern pieces out of a bolt of fabric. The bolt comes off the loom at a **fixed width `W`** — you can't make it any wider — but it's as long as you need, and fabric is sold by the length. So the more length `L` you use, the more you pay and the more you waste.

This is a packing problem, not a jigsaw or tiling problem. The pieces are independent shapes and are not designed to fit together. They may not be able to be arranged into a single clean rectangle, solid block, or complete outer silhouette. Visible gaps, holes, unmatched boundaries, and an irregular overall outline are normal shapes of the pieces.

Note: the stripes printed on the pieces are just decoration — you don't have to line them up with the run of the fabric. You may translate and freely rotate each piece; you may not flip or scale it.

Required Output Format

The examples include labeled outputs so you can infer the task mechanics. In examples, `ground_truth.json` holds the labeled answer (the optimal length).

```
puzzle.png # the input puzzle, copied or referenced for context
assembled.png # your predicted packed layout.  The pieces sit tucked against the top of the fabric; a horizontal line marks L, the length actually consumed -- drawn at the lowest point reached by any piece. Everything below that line is unused fabric and stays blank.
ground_truth.json # the optimal length when building an example
solve.mp4 # generated video illustrating the solving process, should be 81 frames and 480(H)x832(W) resolution.  The first frame must be identical to puzzle.png (pieces scattered, unplaced) and the last frame must be identical to assembled.png (all pieces nested into the fabric). The frames in between animate each piece moving/rotating from its start position into its final packed slot.
```

Prompt A.4. Her-Trees task description
Complete saved task description, engine B.

Task

Task Description

You're handed a few loose brush-stroke cut-outs (`case/puzzle.png` shows the pieces). Each fragment is an organic, irregular brush shape: some visibly change direction within the same piece, forming curves, hooks, corners, or bends. They start out scattered and rotated at random angles, so on their own they look abstract. But they're made to fit together into one thing: a short word, hand-painted in the same loose brush style.

Slide and turn the pieces into their intended places and the word appears — that word is the answer. The strokes are rough and slightly gappy, so you read the word the way you read handwriting: by letting your eye complete the letter shapes. Each letter is built from several stroke segments.

Pieces that share the same color share the same movements. All strokes of one color translate and rotate together as a single rigid group.

Required Output Format

The examples include labeled outputs so you can infer the task mechanics. In examples, `ground_truth.json` holds the hidden answer as text.

```
puzzle.png # Input file
assembled.png # Composed final state
ground_truth.json # the hidden answer as text when building an example
solve.mp4 # generated video illustrating the solving process, should be 81 frames and 480(H)x832(W) resolution.  The first frame must be identical to puzzle.png (pieces scattered, unplaced) and the last frame must be identical to assembled.png (all pieces nested into the fabric). The frames in between animate each piece moving/rotating from its start position into its final packed slot.
```

Prompt A.5. Tangram task description
Complete saved task description, engine C.

Task

Task Description

You are given seven colored Tangram pieces scattered around a single faint target silhouette. Each piece keeps its size and geometry as it moves across the canvas. All pieces must remain flat on the same plane, without stacking or overlapping one another.

Slide and turn the pieces into the target so that they fit together as one complete figure. The colored pieces cover the target while preserving their individual boundaries, and the finished silhouette resembles a familiar object or animal. That object is the answer.

Each case is a distinct arrangement of the same seven primitive Tangram pieces. Do not generate new cases by applying only a global rotation or translation to one master arrangement. Vary the relative positions and orientations of the pieces. The answer identifies the arrangement class, not necessarily a real-world object (You can use like ARRANGMENTS_XXX to describe an ensemble).

The examples show the intended assembly and the motion from the scattered state to the completed figure.

Required Output Format

```
puzzle.png # Input with the pieces scattered around the target
assembled.png # Completed Tangram figure
ground_truth.json # Hidden object name when building an example
solve.mp4 # 81-frame, 832×480 assembly animation
```

The first video frame matches `puzzle.png`, and the last frame matches `assembled.png`. The frames between them show the pieces translating and rotating from their starting positions into the target.

Prompt A.6. RushHour task description
Complete saved task description, engine D.

Task

Task Description

You are given a continuous-space Rush Hour puzzle. Colored rectangular vehicles move on fixed dashed tracks, and the pink/red vehicle labeled **R** must reach the green exit.

Each vehicle may move only forward or backward along its own track. It cannot rotate, resize, deform, leave the track, pass through another object, or cross the board boundary. Only one vehicle moves per action. Every vehicle remains tracked throughout the full video; no vehicle is ignored after it moves. Each vehicle's letter label must remain large, clear, and legible throughout the video.

The red vehicle is not fixed to one starting position or orientation. Its starting side and direction vary across generated puzzles.

Levels 1 through 5 correspond to the minimum number of actions needed to solve the puzzle. Evaluation extracts the ordered action trace from the video and replays it in the world engine. Multiple valid routes may solve the same puzzle.

Required Output Format

```
puzzle.png # Initial puzzle state
assembled.png # Solved final state
ground_truth.json # Ordered solution action list
solve.mp4 # Solution video; vehicles move one at a time; 81-frame,
832x480, 16 FPS H.264 solution animation.
```

The first video frame matches `puzzle.png`; the last matches `assembled.png`. The camera and scene remain fixed throughout the video.

Action Format

`ground_truth.json` contains only the correct ordered route:

```
{
"actions":  ["A backward", "R forward"]
}
```

`forward` follows the vehicle's arrow; `backward` moves against it.

A.3 CONSTRUCTION-AGENT VERSIONS AND TIMING

The principal CLI versions were Claude Code 2.1.216, Codex CLI 0.144.6, and OpenCode 1.18.32. Later TANGRAM and RUSHHOUR Claude runs used 2.1.221; the selected RUSHHOUR Codex run used 0.139.0. The OpenCode version is recorded by the experiment runner.

Recorded wall times were 31.3 min (RUSHHOUR, Claude), 45.4 min (TANGRAM, Claude), and 14.5–23.4 min (four OpenCode runs). These run-level measurements include model calls and local execution.

B WORLDENGINE SKILL AND REVIEW PROTOCOL

The following excerpts document the construction and review procedure supplied to the Claude Tangram builder (`history.jsonl`, record 13).

B.1 WORKFLOW AND ITERATION BUDGETS

Step	Main artifact	Check and next action
1. Infer	`task_logic.md`	State rules, controls, evaluation semantics, visible inputs, and ambiguities. Validate the document contract.
2. Implement	Generator, renderer, evaluator, tests, and examples	Run the build and executable tests. Repair failed checks before review.
3. Critique	`critic_results_run_N.json`	A separate subagent inspects the artifacts, cites evidence, and reports failures and a target.
4. Route	`refinement_feedback.md`	Return to task logic for structural changes or implementation for code/evaluator fixes. Stop successfully only after acceptance.

The loaded skill specifies `MAX_OUTER_ITERS=3` for the construction/review loop and `MAX_RULE_ITERS=6` for hypothesis search when a rule cannot be directly inferred. The accepted Claude Tangram trace records four critic calls, extending the default outer-loop budget; the rejected OpenCode Tangram trace records three.

Prompt B.1. Task interpretation and instance generation
Skill excerpt: Task Input Contract.

Treat `task_description.md` as the authoritative task specification. Read it before inspecting examples. Derive the initial skeleton of the required case format, solver-visible inputs, answer or prediction format, difficulty organization, and evaluation semantics from that file.

Distinguish two cases based on whether you can confidently infer the generative rule:

- **Rule directly inferable.** You can confidently state one generalized rule that fits all the examples. Write it into `## Core Rules`; examples confirm and refine it.
- **Rule must be searched.** You cannot confidently generalize a rule by inspection — the description leaves the mechanic to the examples, as is normal and flexible. The real rule lives only in the labeled examples and must be *derived* — check step 1 for further exploration loop.

You have to adapt to the specific requirements of each task based on the `task_description.md` and examples. Do not assume that every task 1) is visual, 2) has `easy/medium/hard` difficulty tiers, or 3) uses the same artifact names or input filenames as another task unless the task description says so.

Examples are for inferring mechanics, presentation conventions, and edge cases. Generated cases must be new task instances, not copies, crops, recolorings, traces, or near-duplicates of examples — the point is diversity across task-relevant axes. A world engine that only reshuffles or lightly perturbs the references doesn't test generalization, and the eval data it produces won't stress a solver or expose failure modes.

Prompt B.2. Rule-search loop
Skill excerpt: Step 1. Applies when the rule must be searched.

for iter = 1..MAX_RULE_ITERS:

1. Formulate one hypothesis about the transformation rule. - iter 1: use the structured diffs. - later iters: use the specific failures from the previous iteration.

2. Write a transform code that implements the hypothesis. It takes an input (can be raw input or derived intermeidate information from the task) and returns the expected output.

3. Run the transform on ALL labeled example inputs.

4. Compare each output against the expected output; record exact discrepancies (which elements differ, expected vs. actual).

5. All examples match -> rule validated: go write `task_logic.md` (below). Failures remain -> analyze the failure pattern and refine.

Prompt B.3. Implementation requirements
Skill excerpt: Step 2, rules.

- `build.sh` is the main build/test entrypoint.
- `generate_cases.py` exposes seed and difficulty or complexity controls.
- `evaluate_prediction.py` makes evaluation semantics executable.
- Tests are required, including evaluator tests for correct and wrong predictions.
- Generated examples belong under `worldengine/examples/generated/`. YOU SHOULD GENERATE AT LEAST 3 CASES UNDER EACH DIFFICUTLIES.
- Generated examples MUST follow ALL the task-specific output contract derived in Step 1.
- Solver-visible generated inputs must be self-contained. Without complete information in the visible inputs (e.g., missing necessary files or content in the files), the case is effectively unsolvable: a correct solver will fail not because its logic is wrong but because the information it needed was hidden (e.g., kept in ground-truth files, intermediate state, or evaluator internals). Symmetrically, the visible inputs must **not** contain or hint at the gold answer — the answer key, gold output, or solution-revealing metadata must stay in ground-truth files outside the solver's view.

Prompt B.4. Independent critic and dispatch contract
Skill excerpts: Step 3, role and dispatch. Intervening schema paragraph omitted.

Step 3 is performed by an independent critic subagent. The main agent does **not** self-review the artifacts it just built — that defeats the purpose of the review. Instead, dispatch a fresh subagent that reads the build outputs cold and applies the rubric.

[Excerpt omitted: review JSON schema summary.]

Call the `Agent` tool with `subagent_type: general-purpose`, `description: "Step 3 critic for build-world-engine run <N>"`, and a prompt that hands over:

- `RUN_DIR` (absolute path) and the run number `N`.
- Paths to `<RUN_DIR>/task_logic.md`, `<RUN_DIR>/worldengine/`, and `<RUN_DIR>/.venv/bin/python`.
- Paths to `references/rubrics.md`, `references/output_contract.md`, `references/refinement_feedback_schema.md`, and `validators/check_critic_json.py`.
- A pointer to `references/critic_subagent_brief.md` — instruct the critic to read it first; it contains the full role, procedure, rules, and return contract.

The critic is responsible for **writing and self-validating** `critic_results_run_<N>.json` before returning. The main agent does not re-run `check_critic_json.py` after dispatch.

B.2 Critic criteria and routing

The internal critic uses six coarse 0–5 scores: task representation, logic alignment, complexity control, output diversity, format validity, and implementation quality. Its brief requires reading task logic, the generator and evaluator, tests, and multiple generated cases. The review JSON is schema-checked, and the critic grounds each finding in concrete file, test, or artifact evidence.

Prompt B.5. Critic acceptance rules
Critic brief, Hard Rules. Text also present in the saved Tangram tool output.

- **Evidence required.** Every failure entry must cite a concrete file path and (where applicable) a section, line range, or generated-case index. "Looks fine" and "seems reasonable" are not acceptable.
- **File existence is not a pass.** A complete file layout with hollow content is still a failure.
- **Reject shallow generators.** If generated cases are near-duplicates of the references, vary only on cosmetic axes, or are unsolvable from solver-visible information alone, this is an `output_diversity` or solver-visibility failure under `implementation_failures` — flag it and do not pass the run.
- **Reject leaked answers.** If solver-visible inputs embed or directly hint at the gold answer, flag it and do not pass the run.
- **`overall_pass=true` requires zero open failures** in both `structural_failures` and `implementation_failures`. If either list is non-empty, `overall_pass` is `false` and `recommended_next_action.target` is not `stop`.
- **Do not silently redesign.** Your job is to grade and route, not to rewrite `task_logic.md` or `worldengine/`. Structural fixes belong to Step 1 in the next outer iteration; implementation fixes belong to Step 2.

Review target	Required refinement route
`task_logic`	Return to Step 1 for changes to abstraction, rules, or evaluation semantics.
`worldengine`, `evaluator`	Return to Step 2 for code, test, generated-artifact, or evaluator fixes.
`stop`	Allowed only after all step validators pass and the latest critic reports acceptance with no open failures.

The feedback document records the source critic, route, failure category, target file, specific action, validation command, and validation result. At the iteration limit, runs with open failures retain their repair route and rejection status.

Construction-time checking combines executable tests and critic inspection of generated artifacts, including trajectories where the task requires them. Candidate evaluators are task-specific. For example, a word-answer evaluator and a geometric pose evaluator expose different checks. The case study below shows how the independent critic can detect a mismatch between rendered geometry and evaluator behavior even after a build's tests have passed.

C Recorded construction trajectories

These selected case studies document prompts, tool use, critic findings, and subsequent checks from saved construction runs. The boxes identify recorded instructions and outputs.

C.1 Tangram: critic finding, repair, and acceptance

The accepted run is `task3-cld-opus48-v3` (Claude Opus 4.8, high). Its history records a successful skill invocation at records 11–13 and delivery of an original example image through the image-reading tool at records 26–27. Thus this case documents both skill loading and visual access to a supplied example. Four independent critic dispatches occur at records 587, 753, 934, and 1088. The sequence below focuses on the third-to-fourth review transition.

Stage	Recorded evidence
Review request	Critic 3 is asked to independently inspect task logic, source, generated cases, gold/wrong evaluator behavior, determinism, and video-format consistency.
Failure found	A target polygon loses an interior hole during serialization/rendering. The saved gold assembly has IoU 0.9412 against that target, below the evaluator's 0.98 threshold.
Repair	The builder identifies that its area-sum test cannot detect enclosed holes and changes generation to reject holed arrangements. It strengthens regression checks and regenerates examples.
Recheck	The builder reports no holes in a 120-case sweep and gold IoU 1.0 for the previously failing seed. Critic 4 independently reports gold IoU 1.0 for 24 cases, 22 passing tests, and acceptance.

Prompt C.1. Third-round critic dispatch
Recorded builder-to-critic prompt excerpt, history record 934. Skill path normalized.

You are the independent Step 3 critic for outer iteration 3 of the `build-world-engine` skill. You did NOT build these artifacts; review them fresh as a peer's submission, apply the rubric, cite concrete evidence, and reject shallow work. Do not take prior-run conclusions on trust; verify against the current files.

FIRST, read this brief in full and follow its procedure, hard rules, and return contract exactly: <SKILL_DIR>/references/critic_subagent_brief.md

[Excerpt omitted: run paths, artifact list, and prior-change context.]

Independently confirm: cross-process determinism, target is a single connected polygon in all difficulties, solver-visible puzzle.json is sufficient and does NOT leak the gold answer/poses, evaluator scores gold (IoU 1.0 / pass) above a wrong prediction (fail), output contract conformance (puzzle.png/assembled.png 832x480, ground_truth.json, solve.mp4 = 81 frames @ 832x480 24fps, first frame≈puzzle last≈assembled), genuine cross-case diversity, and internal consistency between task_logic.md, code, tests, and README. Hunt for any remaining or new defects.

Prompt C.2. Critic 3: rejected
Recorded critic output excerpt, critic_results_run_003.json.

Interior holes in the assembled tangram silhouette are silently dropped, breaking the core 'gold passes' guarantee. build_arrangement accepts arrangements whose union is a Polygon WITH an interior ring (area==sum still holds, so _arrangement_valid passes). render._ring_px returns only poly.exterior.coords, so the stored puzzle.json target_silhouette and the faint puzzle.png silhouette fill the hole. Reproduced on shipped case examples/generated/easy/case_0002: the model silhouette has 1 interior ring (hole area 0.5 of 8.0), the stored target area is 42499.99 vs the true 39999.99, and evaluating the case's own gold_poses gives iou=0.9412 < 0.98 threshold so passed=False – the correct/gold assembly is REJECTED by its own evaluator.

Prompt C.3. Builder response to the defect
Recorded builder output, history record 1056.

Real bug: my `area == sum` check doesn't catch *enclosed holes* (uncovered interior region — union area still equals piece sum), and `_ring_px` drops interior rings, so the target fills a hole the pieces can't cover → gold fails IoU on ~5% of cases. Fix: reject holed arrangements. Let me also fix the stale docstring.

Prompt C.4. Critic 4: accepted after rechecking
Recorded critic output excerpt, critic_results_run_004.json.

All key invariants were re-verified independently against the current files and hold. (1) Solvability/faithfulness: for all 24 cases (seeds 0-7 x easy/medium/hard) the case's own gold_poses pass evaluate_prediction (IoU=1.0 >= 0.98, no overlaps, full inventory); target_silhouette is a single connected ring with no interior hole in every case (avoid_rectangle + `if union.interiors: continue` confirmed in tangram.build_arrangement, and the pixel target is built via affine_transform of the clean model silhouette in build_case.py, avoiding the old disconnected-MultiPolygon regression).

[Excerpt omitted: further determinism, diversity, evaluator, and format checks.]

Since no structural or implementation failures exist, accept and stop.

The review also checks generated video dimensions, frame count, and first/last-frame correspondence. The repaired defect concerns consistency between target geometry, rendered inputs, and gold evaluation. Saved images show the post-repair artifacts; the quoted critic records document the earlier defect.

C.2 TANGRAM: AN EXECUTABLE BUILD THAT REMAINED REJECTED

The OpenCode run `task3-opencode-gpt55-default-r1` uses GPT-5.5 with default reasoning. The builder loads the construction skill, reads the labeled answers, and inspects example-file metadata through Python. The final critic reviews generated PNG images.

Software validation completes, but the third critic rejects the disconnected, label-ambiguous assemblies and recommends further implementation repair. The final report retains that repair route when the configured outer-iteration budget is exhausted. The disconnected Tangram artifact in Figure 2 shows the build rejected by the third critic.

Source: OpenCode run `history.jsonl`, records 3 (skill), 13–24 (example discovery, labels, and image-size inspection), 66/139/170 (critic dispatches), and 185 (final report), together with its retained session database and `critic_results_run_003.json`. Claude evidence uses the history record numbers above and saved critic JSON files.

D ABLATION AND EVALUATION PROTOCOLS

D.1 CONSTRUCTION-ABLATION CONDITIONS

The Her-Trees ablation uses Codex with GPT-5.5 at high reasoning effort and crosses neutral and diversity-explicit prompts with nested support sets containing 1, 3, or 6 positive examples. It adds a 3-shot no-skill condition and a condition with 6 positive plus 3 explicitly labeled negative examples. Each of the eight conditions has three independent construction runs. The run is the unit of replication.

The 1-shot set contains the original medium example (ZEN). The 3-shot set contains the original GO, ZEN, and FORK examples. The 6-shot set adds three cases from two independently constructed Her-Trees engines. The six-example set therefore combines original and independently generated examples. The negative examples come from a conflicting task in which same-colored fragments can move independently. They are explicitly labeled as negative evidence.

The neutral prompt is Prompt A.1. The complete alternative prompts are reproduced below from the experiment's prompt files. The runner also supplies the absolute-path contract illustrated in Prompt A.2. In the no-skill condition it omits skill installation but retains the stated artifact, testing, seed, and difficulty requirements. This condition evaluates the contribution of the complete skill procedure.

Prompt D.1. Diversity-explicit construction
Complete experiment prompt. Host home path anonymized where present.

Read ~/task/task_description.md. Use the build-world-engine skill to build a faithful world engine from the positive examples under ~/task/examples and write it under ~/out/run-task.

This is a controlled diversity condition. Preserve the task mechanics exactly, including rigid shared motion for fragments of the same color and solver-visible inputs that neither hide necessary information nor leak the answer. Make diversity explicit and controllable rather than cosmetic: expose independent controls for brush family, palette family, word length, fragment/group structure, scatter layout, and rotation severity. Support at least four visually distinct brush families (for example dry brush, calligraphic, rounded marker, and wet ink) while staying within the loose hand-painted brush-stroke task. Generated cases must vary relative geometry and mechanics-relevant configurations; simple recoloring, global rotation, translation, or near-duplication of examples does not count. Generate a balanced demonstration suite covering the supported style families and difficulty levels, and test the controls and evaluator.

Prompt D.2. Diversity with labeled negative examples
Complete experiment prompt. Host home path anonymized where present.

Read ~/task/task_description.md. Use the build-world-engine skill to build a faithful world engine from the positive Her-Trees examples under ~/task/examples and write it under ~/out/run-task.

This is a controlled diversity and distractor-robustness condition. Directories outside ~/task/examples/_negative_task5 are positive Her-Trees examples. The files under ~/task/examples/_negative_task5 are explicitly labeled negative counterexamples from a different task: their same-colored fragments can move independently, which is not the Her-Trees rule. Use them only to document and reject that conflicting mechanic. Do not reproduce their rule.

Preserve rigid shared motion for Her-Trees fragments of the same color and keep solver-visible inputs complete without leaking answers. Expose independent controls for brush family, palette family, word length, fragment/group structure, scatter layout, and rotation severity. Support at least four visually distinct brush families while staying within the loose hand-painted brush-stroke task. Generated cases must vary relative geometry and task-relevant configurations; simple recoloring, global rotation, translation, or near-duplication does not count. Generate a balanced demonstration suite and test the controls and evaluator.

Prompt D.3. No-skill construction
Complete experiment prompt. Host home path anonymized where present.

Read the task description and examples from the absolute paths in TASK_DESCRIPTION_FILE and TASK_DIR. Without using the build-world-engine skill, write task_logic.md under the absolute RUN_DIR and write the engine under the exact absolute WORLD_ENGINE_DIR. The engine must contain README.md, build.sh, scripts/generate_cases.py, scripts/evaluate_prediction.py, tests/test_build.py, tests/test_generation.py, tests/test_evaluator.py, and examples/generated/. Preserve the task output contract, implement deterministic seed and difficulty controls plus an evaluator, include correct-versus-wrong evaluator tests, and generate at least three cases per difficulty. Do not write to ~/task, ~/out, <HOST_HOME>/task, or <HOST_HOME>/out.

The diversity intervention specifies control axes and style families, requests a balanced demonstration suite, and reiterates task rules. The ablation compares this instruction bundle against neutral prompting, alongside the complete-skill comparison.

D.2 Internal criticism and external engine scoring

Internal criticism (Appendix B) guides construction and reports six axes plus acceptance and repair routes. The paper's four-axis scores are external judgments of engine quality. External reviewers assign these scores directly. Two external scoring protocols appear in the archived experiments.

Earlier four-domain construction reviews. The original rubric directs the reviewer to inspect the engine directory and task references, without reading construction task logic, critic JSON, or refinement feedback. It requires running the build and tests, generating fresh cases, and inspecting images and videos. For each dimension, the score is one plus the number of passed checks, ranging from 1 to 5. The overall score is the mean of the four dimensions. The checks below summarize this earlier scoring protocol.

Dimension	Four checks
Fidelity	Same task family and rendering conventions. Compliance with task rules. Correct answer format. Evaluator implements the stated task criterion.
Executability	Build succeeds. Tests actually run and pass. Same seed reproduces outputs across runs. Gold predictions outperform corrupted predictions.
Controllability	Adjustable controls beyond presets. Difficulty increases on a stated measure. Hard cases visibly differ from easy cases. Same-difficulty cases vary meaningfully.
Information validity	Ground truth contains the answer. Puzzle image does not reveal it. Filenames and auxiliary metadata do not leak it. Visible inputs contain the information needed to solve the task.

Each check needs a file citation, command result, numerical observation, or visual observation. The rubric specifies that a failed build is non-viable and receives only fidelity scoring. Earlier review directories retained engine-identifying names.

Source: `stage1-eval-rubrics/eval-rubrics/RUBRICS.md`.

Later four-domain construction reviews. The later Tangram, RushHour, and OpenCode reviews use neutralized artifact directories and a generic anchored 1–5 rubric. This rubric allows task logic and validation reports as evidence and requests direct per-axis scores, evidence lists, hard failures, and a review-pass decision. Fidelity anchors are 5 for faithful mechanics, geometry, transitions, and visual semantics, 3 for a recognizable task with material mismatch, and 1 for a different task or shortcuts. Executability anchors are 5 for working build/generation/evaluation/tests, 3 for repairable gaps, and 1 for inability to generate or evaluate valid cases. Controllability anchors are 5 for demonstrated meaningful controls, 3 for coarse or entangled controls, and 1 for hard-coded or superficial variation. Information-validity anchors are 5 for complete, non-leaking inputs and fair evaluation, 3 for minor leakage or ambiguity, and 1 for exposed answers, missing required information, or an invalid evaluator. Table 1 reports 12 reviewed task–build configurations, including a supplemental Claude RushHour review under the same anchored rubric. Codex performed this supplemental review with known builder identity on a disposable copy of the completed continuous-space build. The build and 25 tests passed; 25 saved/fresh gold traces passed, 75 corrupted traces were rejected, and 40 files matched across separate same-seed runs. The scores are 4/5/4/5 for fidelity/executability/controllability/information validity, with deductions for corridor-template diversity and limited independent controls.

Source: `experiments/construct_rubric.md` and the archived external review JSON files.

Her-Trees ablation review. The ablation uses an anchored 1–5 rubric over the same four named dimensions, specialized to Her-Trees. The review script provides a neutralized artifact directory and a rubric path. That rubric permits task logic, validation reports, and the deterministic audit as evidence.

Prompt D.4. Ablation reviewer invocation
Complete template from judge_completed_runs.py, with paths replaced.

Act as an independent paper reviewer. Read the rubric at <RUBRIC_PATH> and review the neutralized world-engine artifact at <ARTIFACT_DIR>. Read audit.json if present. Do not modify any files. Return only one valid JSON object following the rubric schema, with no Markdown fence.

Prompt D.5. Ablation score anchors
Excerpt: Fidelity and Executability. Other axes follow below.

• 5: mechanics and visual semantics match the task, including rigid shared

motion for same-colored Her-Trees fragments; generated cases are recognizable members of the task family and are not near-copies.

• 3: core task is recognizable but has material visual or mechanics mismatch.

• 1: implements a different task or relies on shortcuts.

Executability

• 5: build, generator, evaluator, tests, and generated artifact contract work

end to end; correct predictions pass and wrong predictions fail.

• 3: engine runs with repairable gaps or incomplete coverage.

• 1: cannot generate or evaluate valid cases.

Prompt D.6. Ablation score anchors and output
Excerpt: Controllability, Information validity, and return contract.

• 5: deterministic seed plus meaningful independent controls for difficulty and task-relevant variation, with tests or generated evidence that controls act.
• 3: seed and coarse difficulty exist but controls are entangled or unverified.
• 1: hard-coded examples or superficial perturbations only.

Information validity

• 5: solver-visible inputs contain everything needed, expose no answer or solution metadata, and the evaluator scores the specified task fairly.
• 3: minor leakage or ambiguity with a mostly usable evaluator.
• 1: answer-bearing inputs, hidden required information, or invalid evaluator.

Return JSON with `fidelity`, `executability`, `controllability`, and `information_validity` numeric scores, an `average`, a short `evidence` list per axis, `hard_failures`, and `review_pass`. Do not infer the experimental condition from directory names and do not reward explicit diversity wording without generated evidence.

D.3 Source-blinded visual preference

The preference experiment uses 30 difficulty-matched initial-image pairs, with five pairs at each of three difficulties for each of Cloth Nesting and Her-Trees. The evaluator is Gemini-3.5-Flash. The pair construction seed is 20260803. Each comparison receives two images in anonymized slots, a task-specific prompt, and a pair identifier. Source mappings and provenance files are withheld until after the response. Each pair is evaluated in a fresh context, and ties are disallowed. The returned fields are `pair_id`, `preferred_example` (1 or 2), and `confidence` (0–1). Confidence records the judge's self-reported certainty.

Prompt D.7. Visual-preference wrapper
Complete runner template. TASK_PROMPT is one of the next two boxes.

Blind pair ID: <PAIR_ID>

Perform one independent blind comparison. The first attached image is Example 1 and the second attached image is Example 2. Do not infer or name either image's generator or provenance. Choose which visible image better represents a valid input puzzle for the task. Ties are not allowed.

<TASK_PROMPT>

Return only the structured JSON requested by the response schema.

Prompt D.8. Cloth Nesting preference prompt
Complete task-specific text from prompts.md. Runner retains its task heading.

You are given two examples. Which example better fits the task description below?

You are a tailor cutting pattern pieces out of a bolt of fabric. The bolt comes off the loom at a **fixed width W** — you can't make it any wider — but it's as long as you need, and fabric is sold by the length. So the more length L you use, the more you pay and the more you waste. You're given a set of pattern pieces — irregular, non-convex shapes with wavy, organic edges — that all have to be cut from this one bolt (`case/puzzle.png` shows the pieces you must place). The pieces are independent cutting patterns and may not connect to form a complete larger shape. Lay every piece out across the width and pack them as tightly as you can so the total length you have to cut, L, is as short as possible. Two pieces may never overlap, and every piece must sit fully inside the fabric. Note: the stripes printed on the pieces are just decoration — you don't have to line them up with the run of the fabric. You may translate and freely rotate each piece; you may not flip or scale it.

Using only visible evidence, choose which image is the better example of a valid input puzzle for this task. Return only the required JSON object.

Prompt D.9. Her-Trees preference prompt
Complete task-specific text from prompts.md. Runner retains its task heading.

You are given two examples. Which example better fits the task description below?

You're handed a few loose brush-stroke cut-outs. Each fragment is an organic, irregular brush shape: some visibly change direction within the same piece, forming curves, hooks, corners, or bends. They start out scattered and rotated at random angles, so on their own they look abstract. But they're made to fit together into one thing: a short word, hand-painted in the same loose brush style.

Slide and turn the pieces into their intended places and the word appears — that word is the answer. The strokes are rough and slightly gappy, so you read the word the way you read handwriting: by letting your eye complete the letter shapes. Each letter is built from several stroke segments.

Pieces that share the same color share the same movements. All strokes of one color translate and rotate together as a single rigid group.

Using only visible evidence, choose which image is the better example of a valid input puzzle for this task. Return only the required JSON object.

The chosen slot is mapped back to its source only after judging. The resulting wins are 13/15 for Cloth Nesting and 5/15 for Her-Trees (18/30 combined). The judged inputs are `puzzle.png` images. Figure 5 presents illustrative rollout frames alongside these initial-image preferences.

Source: `stage1-double-blind/prompts.md`, `run_gemini_blind_test.py` (`build_prompt`), `response_schema.json`, and the pair manifest.

D.4 Construction diversity diagnostics

Diagnostic protocol. We evaluate saved initial-board images from each build: 12 images for diversity-explicit 3- and 6-shot runs and nine for all other runs. Within each build, we compare all unordered image pairs. dHash is the Hamming distance between 64-bit horizontal difference hashes, normalized by 64; hashes use grayscale images resized to 9×8 with Lanczos interpolation. Color is half the L1 distance between normalized 64-bin RGB histograms (four bins per channel), computed after bilinear resizing to 128×74. Dup is the fraction of pairs with dHash Hamming distance at most five bits. We first aggregate pairs within each build, then average the three build-level values per condition; standard deviations are computed across builds. Table 5 reports the diagnostics.

Table 5: **Image-level diagnostics for the Her-Trees construction ablation.** Each condition contains three independent builds. dHash and Color report mean $\pm$ sample standard deviation across builds; Dup reports the mean fraction of near-duplicate image pairs.

Prompting	Ex.	Images/run	dHash	Color	Dup
Neutral, no skill	3	9	.329±.265	.031±.027	.33
Neutral	1	9	.457±.052	.066±.025	.00
Neutral	3	9	.369±.089	.043±.011	.00
Neutral	6	9	.479±.021	.055±.004	.00
Diversity-explicit	1	9	.431±.049	.106±.014	.00
Diversity-explicit	3	12	.430±.091	.079±.013	.00
Diversity-explicit	6	12	.491±.001	.108±.043	.00
Diversity + negative	6+3	9	.431±.057	.065±.037	.00

Negative-example control. At 6-shot, diversity-explicit construction has mean fidelity 3.67; adding labeled negative examples gives fidelity 4.00, average rubric quality 4.50, and 4.33 active axes.

E CURRICULUM CONTROLLERS AND TRAINING DETAILS

E.1 FROM VALIDATION SCORES TO TRAINING EXPOSURES

This case study uses validation feedback to adapt training allocation through the engine's difficulty controls and solution validator. We instantiate two feedback policies using exponential moving averages and scalar Kalman filtering with PID allocation.

Each policy uses 50 controller-validation cases per level and updates the allocation every two epochs, with 2,000 training exposures per epoch and 500–1,500 exposures per level. Fixed maintains a 1:1 allocation.

EMA. We smooth each level's validation solving rate with an exponential moving average and allocate training exposures in proportion to the remaining error.

Kalman. We apply a scalar Kalman filter to each level's validation solving rate and use a PID rule on the estimated L2–L3 performance gap to adjust L3 exposure.

Table 6: Feedback-policy settings. Solving rates and performance gaps are expressed as fractions; allocation increments and change limits are measured in training exposures.

Component	Settings
EMA	Smoothing coefficient 0.5; allocation increment 250; maximum change per update 250.
Kalman filter	Process variance 0.0016; binomial observation variance with floor 0.0025; initial variance 0.01.
PID allocation	Proportional/integral/derivative gains 600/200/300; integral decay 0.7 and bounds ± 0.6.
Kalman–PID allocation	Allocation increment 50; maximum change per update 300.

Training examples are sampled through deterministic shuffled cycles within each level's fixed pool.

Source: Archived implementation: `code/rsi/task4_rsi.py`.

E.2 TRAINING PROTOCOL AND METRIC SCOPE

Experimental setup. We evaluate validation-guided training allocation on engine-generated Rush-Hour instances and assess transfer to a separate reference engine. Fixed, EMA, and Kalman continue from the same Fixed epoch-2 weights for 18 further epochs (3–20), using Wan2.2 TI2V-5B full-DiT fine-tuning at learning rate 10^{-5} with four GPUs and ZeRO-3. Each epoch has 2,000 exposures from an unchanged pool of 1,000 L2 and 1,000 L3 cases. Evaluation uses 832×480 resolution, 81 frames, and 50 inference steps. Controller validation uses 50 cases per level to guide allocation. Cross-engine OOD evaluation uses a separate set of 25 cases per level. We report all nine scheduled evaluations at epochs 4, 6, . . ., 20, with one training trajectory per policy.

Metric scope. The OOD evaluator parses state from the generated first frame and actions from the video. Its validator checks the parsed action plan, including a solved prefix, with maximal-slide transition semantics. Reference actions are used afterward for exact matching. Solving rate uses the validator outcome for the parsed plan. Full-video parse validity and exact-action agreement are recorded as additional diagnostics. Across the nine common checkpoints (450 case–checkpoint evaluations per method), Fixed, EMA, and Kalman have 14, 15, and 17 validator-solved cases with invalid video parsing, respectively. State parsing is valid in 389, 353, and 354 evaluations.

Table 7: Curriculum results. Means average nine scheduled checkpoints, in percent. ID denotes controller validation. OOD intervals use paired bootstrap resampling over cases with epochs grouped (20,000 draws).

Method	ID L3 mean	ID mean	OOD mean	OOD exact	OOD Δ [95% CI]
Fixed	60.9	76.9	29.8	14.2	—
EMA	71.8	83.2	32.0	15.8	+2.2 [−4.4, +8.9]
Kalman	72.9	82.8	32.7	15.3	+2.9 [−2.7, +8.7]

Positive hard-level validation signal. Mean L3 controller-validation success increases from 60.9% with Fixed to 71.8% with EMA and 72.9% with Kalman, gains of 10.9 and 12.0 percentage points (Table 7). Mean overall validation success increases from 76.9% to 83.2% and 82.8%. At epoch 20, overall validation is 87%, 88%, and 81%, and L3 validation is 78%, 78%, and 68%, respectively. Both feedback policies adjust difficulty proportions during training (Figure 4).

Cross-engine evaluation. Mean OOD solving rate is 29.8%, 32.0%, and 32.7% for Fixed, EMA, and Kalman. Paired gains are +2.2 points with 95% CI [−4.4, +8.9] and +2.9 points with CI [−2.7, +8.7]. At epoch 20, OOD success is 30%, 40%, and 42%. Table 9 reports every evaluated checkpoint.

E.3 Complete checkpoint results

Tables 8 and 9 report every common scheduled checkpoint, including per-level outcomes. These per-level trajectories complement the mean comparisons in Table 7.

Table 8: Complete controller-validation checkpoint results (%). L2/L3 lists the two level-specific success rates. All is their balanced average. These ID cases are queried by the controller.

	Fixed		EMA		Kalman	
Epoch	L2/L3	All	L2/L3	All	L2/L3	All
4	82/40	61	92/44	68	86/38	62
6	88/46	67	94/52	73	94/64	79
8	94/64	79	96/84	90	92/78	85
10	96/52	74	88/60	74	94/80	87
12	98/66	82	96/86	91	88/72	80
14	96/72	84	96/80	88	88/88	88
16	94/74	84	96/82	89	98/80	89
18	92/56	74	96/80	88	100/88	94
20	96/78	87	98/78	88	94/68	81

Table 9: Complete cross-engine OOD checkpoint results (%). L2/L3 lists the two level-specific success rates. All is their balanced average. Solving rates use the corrected action-plan validator.

	Fixed		EMA		Kalman	
Epoch	L2/L3	All	L2/L3	All	L2/L3	All
4	20/20	20	40/8	24	44/4	24
6	40/24	32	44/16	30	52/24	38
8	64/12	38	40/16	28	28/40	34
10	28/20	24	36/28	32	24/12	18
12	28/8	18	64/36	50	32/4	18
14	48/16	32	52/16	34	48/24	36
16	52/44	48	32/24	28	52/40	46
18	36/16	26	24/20	22	48/28	38
20	32/28	30	52/28	40	56/28	42

F SUPPLEMENTARY FIGURES AND RESULTS

F.1 CONSTRUCTION VISUALIZATIONS

Figure 5 pairs reference and synthesized rollout frames with the source-blinded initial-image preference results reported in Section 3.2. Figure 6 summarizes the ablation conditions reported in Section 3.3.

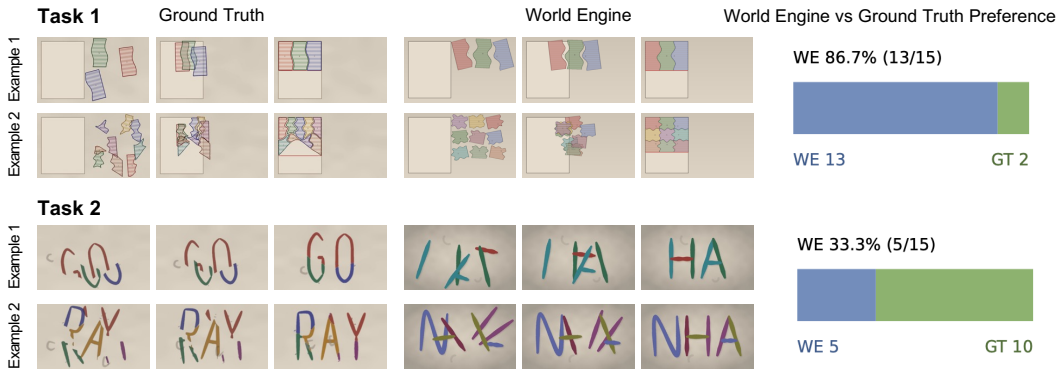

Figure 5: **Visual realism.** Reference and synthesized rollouts (left); blinded preference over 30 pairs (right).



Figure 6: **HER-TREES construction ablation** across 24 runs (three per condition).

F.2 VISUAL PREFERENCE BY DIFFICULTY

Table 10: Blinded sample comparison. Cells are WorldEngine wins / pairs; the reference engine wins the remaining pairs.

Task	Level 1	Level 2	Level 3	Total	WE win rate
Cloth Nesting	4/5	4/5	5/5	13/15	86.7%
Her-Trees	1/5	1/5	3/5	5/15	33.3%
Combined	5/10	5/10	8/10	18/30	60.0%

F.3 OPEN-LOOP OOD-100 EVALUATION

The OOD-100 comparison uses the same 100 reference-engine cases (50 at each of L2 and L3), with one generated video per case and 20 inference steps. The evaluator infers an initial state from the generated first frame and parses the action sequence from the video. It replays the plan under the task rules and checks goal completion. The reported solving rate uses this replay outcome. As a parsing diagnostic, full-video parse validity and plan completion jointly hold for 1/100, 14/100, and 21/100 cases for the base model, Epoch 10, and Epoch 16, respectively. Between Epochs 10 and 16, L2 gains seven solved cases and loses seven, retaining the same total of 15/50.

Table 3 also lists published video-planning results with their source-specific evaluation protocols. The VIPE entries are approximate per-video rates reconstructed from rounded error-reduction plots, using the best of 20 visual variants on 10 puzzles per level and 10 videos per puzzle. The Wan2.2 L3 entry gives the 1–5% range compatible with the rounded plot.

F.4 PARSER REGRESSION ON CLEAN SOLVER VIDEOS

The RushHour engine/parser regression covers 150 clean solver-video cases, 30 at each of five levels. The initial report records 150 parser-valid, 119 solved-plan, and 117 exact-action cases; the later report records 150/150 for all three measures. The original and speed-fixed engine trees contain copies of this same regression sequence.