

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL <https://arxiv.org/abs/2210.03629>.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Appendix

A THE USE OF LARGE LANGUAGE MODELS

We used a large language model (LLM) strictly as a general-purpose writing assistant for surface-level editing, such as grammar correction, wording polish, and minor style consistency. The LLM was not used for conceptual ideation, literature review, algorithm or model design, data collection or labeling, experiment setup, result analysis, drafting of technical content.

B MORE DETAILS ON METHODOLOGY

B.1 METHOD COMPARISONS

Table 5: Comparison of AGENT2WORLD and related approaches. **Feedback** stands for whether the method uses execution/checker/test signals during generation. **External Knowledge** stands for whether explicit web/external knowledge retrieval is a first-class component. **Type** represents environment types, where • and ◦ stand for discrete and continuous environments, respectively.

Method	Representation	Core Paradigm	Feedback	External Knowledge	Environment Type
Text2World (Hu et al., 2025a)	PDDL	Static Workflow	Y	N	•
Guan et al. (2023)	PDDL	Static Workflow	Y	Human	•
Oswald et al. (2024)	PDDL	Static Workflow	Y	N	•
Smirnov et al. (2024)	PDDL	Static Workflow	Y	N	•
AgentGen (Hu et al., 2025b)	PDDL	Static Workflow	Y	N	•
WorldCoder (Tang et al., 2024)	Code	Static Workflow	Y	N	• ◦
GIF-MCTS (Dainese et al., 2024)	Code	Static Workflow	Y	N	• ◦
ByteSized32 Wang et al. (2023)	Code	Static Workflow	Y	N	•
LLM+AL (Ishay & Lee, 2025)	Action Language	Static Workflow	Y	N	•
Direct Generation	PDDL & Code	Direct	N	N	• ◦
AGENT2WORLD _{Single}	PDDL & Code	Adaptive Agent	Y	N	• ◦
AGENT2WORLD _{Multi}	PDDL & Code	Adaptive Agent	Y	Internet	• ◦

As is shown in Table 5, we compare the related methods.

B.2 PER-AGENT TOOL CONFIGURATION

Table 6: Per-agent configuration.

Agent	Tools
Deep Researcher	browser_search; browser_open
Model Developer	file_tool; sandbox; run_code
Simulation Tester	play_env; file_tool
Unit Tester	run_code; run_bash; file_tool

Detailed per-agent tool configuration is presented in Table 6.

B.3 PSEUDO CODE OF AGENT2WORLD_{Multi}

We formalize the process of AGENT2WORLD_{Multi} in Algorithm 1.

Algorithm 1: The execution pipeline of AGENT2WORLD_{Multi}

Input: T, N
Output: e
 $N_r \leftarrow$ predefined integers;
 $R \leftarrow \emptyset$;
 $E \leftarrow \emptyset$;
 $Q \leftarrow \text{ExtractQuestions}(T)$;
for $r \leftarrow 1$ **to** N_r **do**
 $q \leftarrow \text{ResearchAgent}(\text{select}, \{Q, E, R\})$;
 if $q = \emptyset$ **then**
 | **break**
 $L \leftarrow \text{WebSearch}(q)$;
 $E \leftarrow E \cup \text{ResearchAgent}(\text{summarize}, \{L\})$;
 $R \leftarrow \text{ResearchAgent}(\text{update}, \{T, E, R\})$;
 $F_t \leftarrow R$;
 $C_{\text{last}} \leftarrow \emptyset$;
for $n \leftarrow 1$ **to** N **do**
 $C_d \leftarrow \text{DevelopAgent}(T, F_t)$;
 if $C_d \neq \emptyset$ **then**
 | $C_{\text{last}} \leftarrow C_d$;
 | $p_{\text{code}} \leftarrow \text{FileTool}(\text{save}, C_d)$;
 else
 | **continue**
 $C_t \leftarrow \text{UnitTestAgent}(C_d, T, R)$;
 $p_{\text{test}} \leftarrow \text{FileTool}(\text{save}, C_t)$;
 $U_t \leftarrow \text{CodeTool}(\text{run_tests}, \{p_{\text{code}}, p_{\text{test}}\})$;
 $S_t^* \leftarrow \text{PlayEnv}(C_d)$;
 $S_t \leftarrow \text{SimulationTestAgent}(S_t^*, T)$;
 if $U_t.\text{pass} \wedge S_t.\text{pass}$ **then**
 | $e \leftarrow C_{\text{last}}$;
 | **return** e ;
 $F_t \leftarrow \text{MergeFeedback}(U_t, S_t)$;
 $e \leftarrow C_{\text{last}}$;
return e

C MORE DETAILS ON BENCHMARKS

C.1 SIDE-BY-SIDE COMPARISON

Table 7: Overview of Text2World (Hu et al., 2025a), Code World Models Benchmark (CWMB) (Dainese et al., 2024), and ByteSized32 (Wang et al., 2023). “Type” denotes the target representation (PDDL vs. executable code). Metrics are shown at the family level. A detailed explanation of each metrics is presented in Appendix C.2.

Benchmark	#Environments	Type	Metrics (core)
Text2World	103	PDDL	Executability; Domain similarity; F1 scores
CWMB	18	Code (Python)	Accuracy; Normalized return \mathcal{R} (discrete/continuous)
ByteSized32	32	Code (Python)	Technical validity; Specification compliance; Winnability; Physical reality alignment

A side-by-side comparison of the evaluated benchmarks in this paper is presented in Table 7

C.2 METRIC EXPLANATION

Text2World

Executability. *Name:* *Exec.* *Range:* $[0, 1]$ (higher is better). Whether the generated {domain, problem} can be successfully parsed and validated by standard PDDL validators; reported as the fraction (percentage) over all test cases. Fine-grained metrics below are computed only when *Exec* = 1.

Domain similarity. *Name:* *Sim.* *Range:* $[0, 1]$ (higher is better). Textual/structural similarity between the generated and gold PDDL measured by a *normalized Levenshtein ratio*.

Let X and Y be the character sequences of the two files with lengths $|X|$ and $|Y|$, and let $\text{Lev}(X, Y)$ denote their Levenshtein distance, then

$$\text{Sim}(X, Y) = 1 - \frac{\text{Lev}(X, Y)}{\max\{|X|, |Y|\}} \in [0, 1]. \quad (1)$$

F1 scores. *Range:* $[0, 1]$ (higher is better). When *Exec* = 1, we parse both generated and gold PDDL into structured representations and report *macro-averaged* F1 for the following components: **Predicates** ($F1_{\text{PRED}}$), **Parameters** ($F1_{\text{PARAM}}$), **Preconditions** ($F1_{\text{PRECOND}}$), and **Effects** ($F1_{\text{EFF}}$). We use the standard definition of F_1 , where P and R denote precision and recall, respectively:

$$F_1 = \frac{2PR}{P+R}$$

CWMB

Prediction Accuracy. *Symbol:* Acc_{pred} . *Range:* $[0, 1]$ (higher is better). *Definition:* We use the same accuracy metric as in the evaluation phase of GIF-MCTS (Sec. 4). Given a validation set $D = \{(s_i, a_i, r_i, s'_i, d_i)\}_{i=1}^N$ and CWM predictions $(\hat{s}'_i, \hat{r}_i, \hat{d}_i) = \text{CWM.step}(s_i, a_i)$, the accuracy uniformly weights next state, reward, and termination:

$$\text{Acc}_{\text{pred}} = \frac{1}{N} \sum_{i=1}^N \left[\frac{1}{3} \mathbf{1}(\hat{s}'_i = s'_i) + \frac{1}{3} \mathbf{1}(\hat{r}_i = r_i) + \frac{1}{3} \mathbf{1}(\hat{d}_i = d_i) \right]. \quad (2)$$

Normalized Return. *Symbol:* \mathcal{R} . *Range:* unbounded (higher is better; $\mathcal{R} > 0$ means better than random; $\mathcal{R} \rightarrow 1$ approaches the oracle). *Definition:*

$$\mathcal{R} = \frac{R(\pi_{\text{CWM}}) - R(\pi_{\text{rand}})}{R(\pi_{\text{true}}) - R(\pi_{\text{rand}})}, \quad (3)$$

where $R(\pi)$ denotes the return. *Protocol:* as in the original setup, we use vanilla MCTS for discrete action spaces and CEM for continuous action spaces; $R(\cdot)$ is averaged across a fixed number of episodes per environment (10 in the original), and $R(\pi_{\text{rand}})$ uses the environment's random policy baseline.

ByteSized32

Technical Validity. *Range:* $[0, 1]$. Measured in the order of API calls, such that failure of an earlier function implies failure of subsequent tests. `Game initialization` is evaluated once at the beginning of the game, whereas `GENERATEPOSSIBLEACTIONS()` and `STEP()` are evaluated at *every step*. We check:

- *Game initialization:* the game/world initializes without errors;
- *Valid actions generation:* the routine that enumerates valid actions for the current state returns without errors (verified via a bounded path crawl);
- *Runnable game:* a bounded-depth crawl of trajectories executes without errors.

Specification Compliance. *Range:* $[0, 1]$. An LLM acts as the judge for *true/false* compliance against the task specification. The prompt provides the task spec $\{\text{GAME_SPEC}\}$, the game code $\{\text{GAME_CODE}\}$, and an evaluation question $\{\text{EVAL_QUESTION}\}$; the LLM is instructed to first output `Yes/No` and then a brief rationale. To reduce variance, we use a fixed prompt template and perform multiple independent runs with majority vote/mean aggregation. We report three submeasures: *Task-critical objects*, *Task-critical actions*, and *Distractors*.

Physical Reality Alignment. Range: $[0, 1]$. Automatic evaluation proceeds in two stages:

(1) *Trajectory generation*: perform a breadth-first crawl using the action strings returned by `GENERATEPOSSIBLEACTIONS()` at each step; actions are grouped by verb (first token) and expanded in a bounded manner. If an error occurs, the error message is recorded as the observation and the search continues.

(2) *Sampling and judgment*: group paths by the last action verb, draw a fixed-size subsample approximately balanced across groups, and submit each path—together with the task description `{GAME_TASK}`—to an LLM for a binary judgment (yes/no; errors are treated as failures). The final score is the fraction judged aligned.

Winnability. Range: $[0, 1]$. A text-game agent (LLM agent) attempts to reach a terminal win within horizon H ; we report the fraction of tasks deemed winnable. Given the limited agreement between automatic and human assessments for this metric, we prioritize human evaluation in the main results and use the automatic estimate as auxiliary reference.

D MORE DETAILS ON ABLATION STUDY

Table 8: Ablation Study of AGENT2WORLD on CWMB (Dainese et al., 2024).

Method	Discrete Action Space		Continuous Action Space		Overall	
	Accuracy (\uparrow)	\mathcal{R} (\uparrow)	Accuracy (\uparrow)	\mathcal{R} (\uparrow)	Accuracy (\uparrow)	\mathcal{R} (\uparrow)
AGENT2WORLD	0.9174	0.8333	0.3575	0.3050	0.5441	0.4811
No Deep Researcher	0.8794 _{-0.0380}	0.4407 _{-0.3926}	0.3404 _{-0.0171}	0.2201 _{-0.0849}	0.5201 _{-0.0240}	0.2936 _{-0.1875}
No Simulation Tester	0.8920 _{-0.0254}	0.5718 _{-0.2615}	0.3288 _{-0.0287}	0.1577 _{-0.1473}	0.5275 _{-0.0166}	0.3039 _{-0.1772}
No Unit Tester	0.6166 _{-0.3008}	0.3863 _{-0.4470}	0.3025 _{-0.0550}	0.1704 _{-0.1346}	0.4072 _{-0.1369}	0.2423 _{-0.2388}

Detailed experimental results of the ablation study are presented in Table 8.

E MORE DETAILS ON ERROR ANALYSIS

E.1 ERROR ANALYSIS ON TEXT2WORLD AND BYTESIZED32

We visualize the error patterns during evaluation-driven refinement on Text2World and ByteSized32 in Figure 8 and Figure 9.

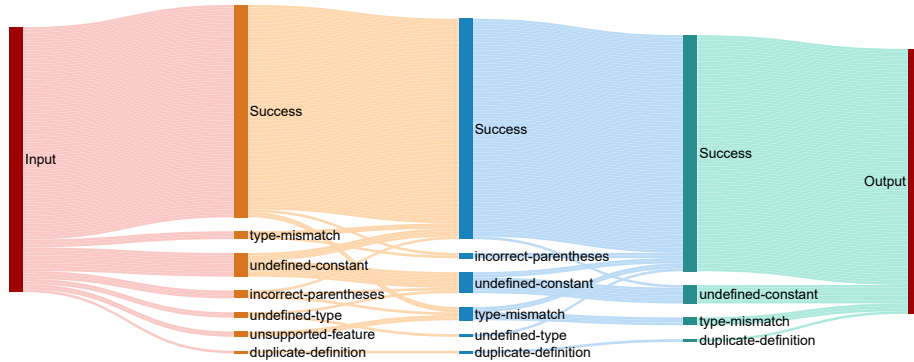


Figure 8: Error distribution of AGENT2WORLD_{Multi} on Text2World.

E.2 DISTRIBUTION OF ERROR TYPES

A detailed proportion of error types on Text2World, CWMB, ByteSized32 are presented in Table 9, Table 10 and Table 11, respectively.

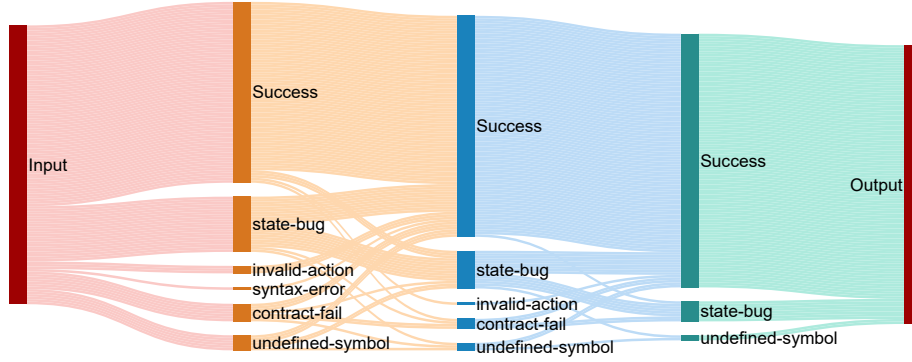
Figure 9: Error distribution of AGENT2WORLD_{Multi} on ByteSized32.

Table 9: Distribution of Syntax Errors in Text2World Across Turns

Error Type	Explanation	Turn 0 (%)	Turn 1 (%)	Turn 2 (%)
undefined-constant	Reference to undeclared constants in predicates or actions.	8.91	7.92	6.93
type-mismatch	Parameter type conflict with declared type constraints.	2.97	4.95	2.97
incorrect-parentheses	Invalid or mismatched parentheses.	2.97	1.98	0.00
undefined-type	Undeclared parent type in hierarchical type definitions.	1.98	0.99	0.00
unsupported-feature	Parser-incompatible features (e.g., either types).	1.98	0.00	0.00
duplicate-definition	Multiple declarations of identical domain elements.	0.99	0.99	0.99

Table 10: Distribution of Syntax Errors in CWMB Across Turns

Error Type	Explanation	Turn 0 (%)	Turn 1 (%)	Turn 2 (%)	Turn 3 (%)
signature-mismatch	Arity/types do not match the declared signature.	27.78	11.11	11.11	5.56
schema-mismatch	Value type/shape/dtype violates the expected schema.	22.22	5.56	0.00	0.00
dynamics-error	State or reward deviates from expected dynamics.	11.11	44.44	33.33	11.11
non-deterministic	Results are inconsistent under fixed conditions.	11.11	0.00	0.00	0.00
judgment-bug	Environment setup inconsistent with the description.	11.11	5.56	11.11	5.56
invariant-violation	Internal invariants are broken (e.g., illegal config).	0.00	5.56	0.00	5.56

Table 11: Distribution of Syntax Errors in Bytesized32 Across Turns

Error Type	Explanation	Turn 0 (%)	Turn 1 (%)	Turn 2 (%)
state-bug	State inconsistency across steps.	19.82	13.51	7.21
contract-fail	Task/API contract not satisfied.	6.31	3.60	0.00
undefined-symbol	Reference to undeclared name/type/domain/constant.	5.41	2.70	1.80
invalid-action	Unknown or unsupported action not safely handled.	2.70	0.90	0.00
syntax-error	Load/parse failure (e.g., null bytes, syntax/indentation errors).	0.90	0.00	0.00

F PROMPT EXAMPLES

Deep Researcher

You are a world-class Systems Analyst and Technical Specification Writer, specializing in creating reinforcement learning environments compliant with the Gymnasium API. Your mission is to transform an ambiguous task description into a precise, actionable, and verifiable technical specification.

<Environment Name>

CliffWalking-v0

</Environment Name>

<TASK DESCRIPTION>

Cliff walking involves crossing a gridworld from start to goal while avoiding falling off a cliff.

Description ...

</TASK DESCRIPTION>

<Workflow>

Please strictly follow the following six-step process:

- **Deconstruction and Analysis (Use Version Locking)**
 - Identify all ambiguities, gaps, and conflicts in the task description.
 - Lock the exact environment version and all key library versions (record name, version, and source link).
 - Categorize gaps by type: missing value/unit/range boundary/time- sensitive/ambiguous reference/unclosed list/conflict/no provenance.
- **Planning and Investigation (Authoritative Search + Evidence Log)**
 - For each high/mid-level project, write 1–2 focused queries that include: synonyms/abbreviations, site filters, authoritative domains (e.g., site:numpy.org, site:mujoco.readthedocs.io, site:doi.org), and recency windows (e.g., after 2024-01-01 or “last 2 years”).
 - Execute the query using browser_search and open >= 2 trusted results with browser_open.
 - If the top source disagrees, open >= 1 additional authoritative sources and triangulate.
 - Create an evidence log entry for each opened page: Title | Organization/Author | Version/Submission | URL (+ archived URL) | Publication Date | Access Date (Asia/Singapore) | 3 Key Facts | Confidence (High/Medium/Low).
- **Synthesis and Citation (Conflict Resolution)**
 - Integrate the findings into a concise evidence summary with citations.
 - When sources conflict, explain the differences and justify the chosen resolution (related to version locking).
- **Refinement and Improvement (Specification Patch)**
 - Generate a structured “diff”: action/observation space; rewards; termination/truncation; timing (dt/frame_skip); seeding and certainty; numerical tolerances; dependencies; interface flags.
- **Formalization and Finalization (Ready-to-Use Specification)**
 - Write the final specification according to the <Output Format> , including the public API, core logic, usage scenarios, and a verification plan aligned with metrics and statistical validation.
- **Review and Self-Correction (Compliance Check)**
 - Verify conformance to the <Output Constraints> (OUTPUT_CONSTRAINTS>), version consistency, SI units, ISO dates, and the inclusion of any code.

</Workflow>

<OUTPUT_CONSTRAINTS>

- Strictly adhere to the structure defined in <PLANNING_STRUCTURE>.
- Do **NOT** output runnable code definitions (classes, functions). Only may include short illustrative snippets or pseudo-code.
- All claims about industry standards or common practices **MUST** be supported by citations.
- Use ISO-8601 dates (e.g., 2025-09-02).
- Use SI units for physical and mathematical quantities.
- Data-leakage rule: Do not access, copy, quote, or derive from raw source code in the OpenAI/Gym/Gymnasium repositories or similar code repositories. Do not include any repository code in the output. Prefer official documentation, standards, papers, or reputable secondary sources. If the only available evidence is a code repository, summarize behavior without copying code and mark it as an inference with risks.

<PLANNING_STRUCTURE>

- Your output must begin with this planning and analysis section.
- **Ambiguity Analysis**
 - List each ambiguity/vagueness/conflict and mark Impact: High / Medium / Low.
 - Cover at least: missing numeric value, missing unit, missing boundary/range, time-sensitive items, unclear references, open lists (“etc.”/“e.g.”), conflicts, and missing citation.
- **Investigation Plan**
 - For each High/Medium item, provide one atomic question.
 - For each question, provide 1–2 executable queries including: synonyms/abbreviations, a site filter to authoritative domains, and a time window (e.g., `after:2024-01-01` or “past 2 years”).
 - State the minimum evidence policy: High/Medium $\rightarrow \geq 2$ credible sources; if disagreement \rightarrow add ≥ 1 more for triangulation.

<Formula requirements>

- For any formula, define all symbols, units, and applicability constraints.
- Cite the source of the formula immediately after its definition.
- Provide the complete formula rather than a descriptive explanation.

<OUTPUT_FORMAT>

- Please provide the final specification document structured as follows. This is the primary deliverable. Do **NOT** include code.
- <Version & Provenance>
- <Evidence Summary>
- <Spec Patch>
- <Theoretical Foundations>
- <Final Specification>
- <Assumptions & Risks>
- <Third-Party Library Usage>

Model Developer

<TASK DESCRIPTION>

Cliff walking involves crossing a gridworld from start to goal while avoiding falling off a cliff.

Description ...

</TASK DESCRIPTION>

<Research Report>**# CliffWalking-v0 Environment Specification****Version & Provenance**

- Environment: CliffWalking-v0
- Gymnasium version: v0.26.3 (2022-09-15) and latest documentation snapshot (2025-01-01)
- Scope: This specification normalizes the CliffWalking environment as implemented in Gymnasium v0.26.3, aligning with the classic gridworld cliff walking task.
- Accessed date: 2024-06-01 (Asia/Singapore)

Evidence Summary

- The environment is a 4x12 grid with 48 discrete states indexed by row-major flattening: $\text{state} = \text{row} * 12 + \text{col}$.
- Valid states exclude cliff cells ([3,1..10]) and the goal ([3,11]). The player can occupy all cells in the first 3 rows plus the bottom-left cell [3,0].
- The start state is 36 (row=3, col=0), and the goal state is 47 (row=3, col=11).
- The action space is Discrete(4) with actions: 0 (up), 1 (right), 2 (down), 3 (left).
- Each step yields a reward of -1. Stepping into the cliff yields a reward of -100 and resets the player to the start state; the episode continues.
- The episode terminates only when the player reaches the goal state.
- The `set_state` method is not explicitly documented but is inferred to accept any valid non-terminal, non-cliff state and sets the environment to not done.
- No explicit error handling for invalid actions is documented; valid inputs are assumed.
- The environment is deterministic.

Spec Patch

- `action.space = Discrete(4)`, values {0,1,2,3}, shape (1,)
- `observation.space = Discrete(48)`, integer in [0,47], representing flattened grid index: $\text{row} * 12 + \text{col}$
- `valid states = all grid cells except cliff cells ([3,1..10]) and goal ([3,11])`
- `start_state = 36` (row=3, col=0)
- `goal_state = 47` (row=3, col=11)
- `reward.step = -1` per step
- `reward.cliff = -100` on stepping into cliff
- `episode termination = True` if `state == goal_state`; else `False`
- stepping into cliff resets player to `start_state`, episode continues
- `set_state(state: int)` sets environment state to given valid state, `done = False`
- no maximum episode length or truncation condition
- no explicit error handling for invalid actions; assume valid input
- environment deterministic transitions

Theoretical Foundations**• MDP Definition:**

- State space $S = \{0, \dots, 47\} \setminus \{\text{cliff states, goal state}\}$.
- Action space $A = \{0, 1, 2, 3\}$.
- Transition function $T(s, a, s')$ (deterministic):
 - * If s' is a cliff cell, next state = `start_state`.
 - * Else next state = $s + \delta(a)$ (with grid boundary checks).

- Reward function $R(s, a, s')$:
 - * $R = -100$ if s' is a cliff cell.
 - * $R = -1$ otherwise.
- Episode ends when $s' = \text{goal_state}$.
- **Symbol Table:**
 - s : state (int), flattened grid index, $0 \leq s \leq 47$, s not in cliff or goal.
 - a : action (int), in $\{0 : \text{up}, 1 : \text{right}, 2 : \text{down}, 3 : \text{left}\}$.
 - r : reward (float), -100 or -1 .
 - done: boolean, True if $s = \text{goal_state}$.
 - start_state: 36 (int).
 - goal_state: 47 (int).
 - cliff_cells: set of ints corresponding to grid positions $[3, 1..10]$.
 - nrows: 4 (int).
 - ncols: 12 (int).
- **Assumptions:**
 - No shaping rewards.
 - Deterministic environment.
 - No truncation or max step limit.

Final Specification

- **Environment Name**
 - CliffWalking-v0
- **Public API**
 - `__init__()`
 - `set_state(state: int)`
 - `step(action: int) → (observation: int, reward: float, done: bool)`
- **Core Logic Description**
 - `__init__`:
 - * Initialize grid size (4 rows \times 12 columns).
 - * Define start_state = 36 (row=3, col=0).
 - * Define goal_state = 47 (row=3, col=11).
 - * Define cliff cells as positions $[3, 1..10]$.
 - * Initialize current state to start_state.
 - * Initialize done flag to False.
 - `set_state(state: int)`:
 - * Validate that state is a valid observation (not cliff or goal).
 - * Set current state to state.
 - * Set done flag to False.
 - `step(action: int) → (obs, reward, done)`:
 - * Validate action is in $\{0, 1, 2, 3\}$.
 - * Compute next position based on action with boundary checks:
 - 0: move up (row -1)
 - 1: move right (col $+1$)
 - 2: move down (row $+1$)
 - 3: move left (col -1)
 - * If next position is outside grid, remain in current position.
 - * If next position is a cliff cell:
 - reward = -100
 - next state = start_state
 - done = False

- * Else if next position is goal_state:
 - reward = -1
 - done = True
 - next state = goal_state
- * Else:
 - reward = -1
 - done = False
 - next state = computed position
- * Update current state to next state and return (next state, reward, done).

• Usage Scenarios

– Scenario 1: Normal move into cliff

- * Initial state: 36 ([3,0])
- * Action: 1 (right)
- * Next state: 37 ([3,1]) is cliff → reset to 36
- * Reward: -100
- * done: False

– Scenario 2: Move to goal

- * Current state: 46 ([3,10])
- * Action: 1 (right)
- * Next state: 47 ([3,11]) goal
- * Reward: -1
- * done: True

– Scenario 3: Move up from start

- * Current state: 36 ([3,0])
- * Action: 0 (up)
- * Next state: 24 ([2,0])
- * Reward: -1
- * done: False

Assumptions & Risks

- No truncation or maximum episode length specified; risk of infinite episodes if goal not reached.
- set_state accepts only valid non-terminal, non-cliff states; no explicit error handling for invalid states.
- No explicit error handling for invalid actions; assumed caller responsibility.
- Reward and reset behavior on cliff stepping inferred from documentation; no episode termination on cliff.
- Observation space includes all grid cells but player never occupies cliff or goal states.
- Potential future Gymnasium version changes may alter behavior.

Third-Party Library Usage

- Gymnasium API conventions inform environment interface: discrete action and observation spaces, step and reset semantics.
- No new runtime dependencies introduced.
- RNG seeding not specified; environment deterministic.
- No source code copied; specification based on official Gymnasium documentation and Sutton & Barto RL book.
- Numpy indexing conventions used for grid flattening.

</Research Report>

<Workflow>

1. **Deconstruct Specification:** Carefully review the <Task Describe>,<Research Report>

to fully understand the environment’s specification, including state/action spaces, dynamics, reward function, and termination conditions.

2. **Physics Engine Selection:** Evaluate if the task requires physics simulation. If so, choose an appropriate physics engine for the specific task requirements.

3. **Model Design:** If using a physics engine, design the model structure and embed it as needed in the Python file.

4. **Plan Class Structure:** Outline the ‘Environment’ class, including its internal state variables, helper methods, and the public interface (‘__init__’, ‘reset’, ‘set_state’, ‘step’).

5. **Implement Complete Code:** Write the full implementation of the ‘Environment’ class. 6. **Self-Correction Review:** Meticulously check that the generated code fully complies with the <TASK DESCRIPTION>, the <Research Report>, and all <ImplementationRequirements>.

7. **Finalize Output:** Present the complete, reviewed, and runnable single-file code in the specified final format.

</Workflow>

<ImplementationRequirements>

1. Interface (single file):

- Implement a complete, self-contained Python class Environment with:
 - __init__(self, seed: int | None = None)
 - reset(self, seed: int | None = None) → ndarray (reinitialize the episode and return the initial observation in canonical shape)
 - set_state(self, state) (must accept ndarray or list/tuple in canonical shape)
 - step(self, action) → tuple[ndarray, float, bool] (returns: observation, reward, done)
- Requirements:
 - Single-file constraint: all code, including any model definitions, must be contained in one Python file.
 - For physics-based environments, embed model definitions as string constants within the class.
 - Explicitly define state, action, and observation spaces (types, shapes, ranges, formats).
 - Provide reproducibility (seeding) via the constructor and/or a seed(int) method.
 - Be robust to common representations:
 - * set_state: accept list/tuple/ndarray of the same logical content.
 - * step: accept int / NumPy integer scalar / 0-D or 1-D len-1 ndarray (convert to canonical form; raise clear TypeError/ValueError on invalid inputs).
 - No dependence on external RL frameworks; no Gym inheritance.
 - No external file dependencies (model definitions must be embedded).
 - Maintain internal state consistency; allow reconstruction from observations where applicable.
 - Clean, readable code suitable for RL experimentation.

2. Determinism & validation:

- Provide reproducibility via seed (constructor and/or seed(int) method).
- Normalize inputs: accept equivalent representations (e.g., NumPy scalar/int/len-1 array) and convert to a canonical form.
- Validate inputs; raise clear one-line errors (ValueError/TypeError) on invalid shapes or ranges.

3. Dynamics (MCTS/control oriented):

- For physics-based tasks, prefer suitable physics simulation methods with embedded model definitions over custom physics implementations.
- Choose and document an integration scheme (e.g., implicit integrator, explicit Euler) consistent with the research report.

- Use a stable time step dt ; clamp to safety bounds; keep all values finite (no NaN/Inf).
- Keep per-step computation efficient and allocation-light.

4. Dependencies & style:

- No Gym inheritance or external RL frameworks unless explicitly allowed.
- Allowed: third-party libraries as needed (e.g., NumPy, physics engines, SciPy, Numba, JAX, PyTorch, etc.).
- For robotics/physics tasks, physics engines with embedded model definitions are recommended over custom implementations.
- Clean, readable code suitable for RL experimentation.
- All dependencies must be importable standard libraries or commonly available packages.

</ImplementationRequirements>

<Output Format>

<final> `<code_file_path>` The entrypoint file path of the generated code. **</code_file_path>**

<entrypoint_code> “python # Your complete, runnable single-file implementation here. “

</entrypoint_code> **</final>**

</Output Format>

Unit Tester

<TASK DESCRIPTION> Cliff walking involves crossing a gridworld from start to goal while avoiding falling off a cliff.

Description ...

</TASK DESCRIPTION>

<CodeArtifact path="environment.py"> {code} **</CodeArtifact>**

<ExecutionPolicy>

- Do not modify the student’s source file.
- Create exactly one pytest file at “tests/test_env.py” using file_tool(“save”).
- Import the module from “environment.py” via importlib (spec_from_file_location + module_from_spec).
- Run tests with code_tool(“run”, “pytest -q”); capture exit_code, duration, and stdout/stderr tail.

</ExecutionPolicy>

<TestPlan>

- Sanity: `class Environment` can be imported and instantiated, e.g., `Environment(seed=0)`.
- Contract:
 1. `set_state` accepts list/tuple/ndarray of the same logical content (convert to canonical).
 2. `step(action)` returns a 3-tuple: (observation, reward, done) with expected types/shapes.
 3. Determinism: with the same seed and same initial state, the first step with the same action yields identical outputs.
 4. Action space validation: actions within bounds are accepted, out-of-bounds actions are handled gracefully.
 5. Observation space validation: observations match declared space bounds and shapes.
 6. State space consistency: internal state dimensions match expected environment specifications.
- Acceptance: success iff `pytest exit_code == 0` (all tests pass).

</TestPlan>

<ReportingGuidelines>

- Summarize pytest results in 2–4 sentences; mention the first failing nodeid/assert if any.
- Provide a brief contract coverage assessment and the most probable root cause for failures.
- If failing, add 1–3 concise actionable fixes (no long logs).

</ReportingGuidelines>

<OutputFormat> Return exactly one **<final>** block containing a single JSON object that matches PytestReport: {

"success": true/false,

"analysis": "<2–4 sentence summary/diagnosis>",

"suggest_fix": "1–3 bullets with minimal actionable changes"

} No extra text outside **<final>**. No additional code fences.

<final> { "success": false, "code_result": "", "analysis": "", "suggest_fix": "" } **</final>**

</OutputFormat>

Simulation Tester

Your task is to interact with the environment code and then analyze the feedback from the interaction and propose modifications

<TASK DESCRIPTION>

Cliff walking involves crossing a gridworld from start to goal while avoiding falling off a cliff.

Description ...

</TASK DESCRIPTION>

<CodeArtifact path="environment.py">

{code}

</CodeArtifact>

<ExecutionPolicy>

- Use the play_env tool exactly once on "environment.py" - If the tool throws or cannot run, perform diagnosis from static review only; still produce output in the required format.

</ExecutionPolicy>

<Rubric>

Success (boolean) must be decided from the available signals with graceful degradation:

- **Primary (step-level signals present):**
 - success = true iff the run finished without exceptions AND there is NO misclassified_transition with (valid == false OR state_matches == false).
 - If only observation deltas are available, use obs_matches instead of state_matches.
 - When numeric deltas are provided, treat matches = true if $\max_abs_error \leq 10^{-3}$ or $rel_error \leq 10^{-3}$.
- **Secondary (no per-step signals):**
 - If success_rate exists: success = true iff no exceptions AND success_rate ≥ 0.95 .
 - Else: success = true iff no exceptions AND no invariant/contract violations you can substantiate from code and logs.
- **Reward/termination:**
 - If reward_matches == true AND done_matches == true, explicitly state they match and DO NOT propose changes to reward or termination logic.
- **Action space consistency (discrete & continuous):**
 - If GT exposes Box(low, high, shape): align predicted bounds and expose them (e.g., env.action_space or a getter). Never place clipping inside the integrator; clamp only at action ingestion or at observation.

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

1361

1362

1363

1364

1365

1366

1367

1368

1369

1370

1371

1372

1373

1374

1375

1376

1377

1378

1379

1380

1381

1382

1383

1384

1385

1386

1387

1388

1389

1390

1391

1392

1393

1394

1395

1396

1397

1398

1399

1400

1401

1402

1403

- If GT exposes `Discrete(n)`: actions must be integer indices in $[0, n-1]$; expose `n` (e.g., `gym.spaces.Discrete(n)`); if indices map to continuous commands/torques, list the mapping table and align it with GT; never float-clip discrete actions.
- If action-space info is missing, skip these checks (do not speculate).
- **Internal vs observation:**
 - If clipping or angle normalization is found inside the integrator step (e.g., in `_rk4_step`), this likely causes trajectory drift; propose moving them to the observation path (e.g., `_get_observation`) unless GT specifies otherwise.
 - If latent state is unavailable but observations exist, compare observations instead and state this explicitly.
- **Integrator & timestep:**
 - Mismatches in integrator method (e.g., RK4 vs Euler) or `dt` can cause state divergence even when reward/done match; acknowledge and, if state mismatches persist, propose aligning method/`dt` to GT.
- **Batched/multiple transitions:**
 - If multiple transitions are reported, aggregate sensibly (e.g., mean success_rate or fraction matched ≥ 0.95) before deciding success.

</Rubric>

<Procedure>

1) Static review: scan for action bounds, clipping/normalization inside integrator, integrator/`dt` choice, and how observation is formed.

2) Execute: call `play_env` once.

3) Diagnose: reconcile `play_env` signals with code; if reward/done matched, explicitly say so. If state mismatched, point to ONE OR TWO most likely roots.

4) Suggest: 1–3 smallest patches that directly address the identified root causes.

</Procedure>

<OutputFormat>

Return exactly one <final> block containing a single JSON object that matches PlayReport: { "success": true/false, "analysis": "<2–4 sentences summarizing what happened and why; mention matches/mismatches explicitly>", "suggest_fix": "- bullet 1\n- bullet 2\n- bullet 3 (optional)" } No extra text outside <final>. No additional code fences.

</OutputFormat>

G EXAMPLES OF DEEP RESEARCHER REPORT

Deep Researcher's Report

<Version & Provenance>

- Environment: Pusher-v4
- MuJoCo version: $\geq 2.1.3$ and $< 3.0.0$
- Source release date: 2024-01-01 (approximate for v4)
- Accessed date: 2025-06-01 (Asia/Singapore)

<Spec Patch>

- **action.space** = `Box(low=-2.0, high=2.0, shape=(7,), dtype=np.float32)`
- **observation.space**: `Box` with `low = - ∞` , `high = ∞` , `shape (23,)`, `dtype np.float64`
- **reward.weights** = `{dist: 1.0, ctrl: 0.1, near: 0.5}`
- **reward.formula**: `reward = reward_dist + 0.1reward_ctrl + 0.5reward_near`

- `reward_ctrl` = $-\|\mathbf{a}\|_2^2$ (negative squared Euclidean norm)
- `reward_dist` = $-\|\mathbf{o} - \mathbf{g}\|_2$ (negative Euclidean distance)
- `reward_near` = $-\|\mathbf{f} - \mathbf{o}\|_2$ (negative Euclidean distance)
- `episode.max_steps` = 100
- `episode.termination`: never (terminated always False)
- `episode.truncation`: True if step count \geq max_steps or any observation value is non-finite
- $dt = 0.05\text{s}$ (frame_skip= 5, frame_duration= 0.01)
- `initial_state.joint_positions` = $\mathbf{0}_7$
- `initial_state.joint_velocities`: each joint $\sim U(-0.005, 0.005)$
- `initial_state.object_position`: $x \sim U(-0.3, 0)$, $y \sim U(-0.2, 0.2)$; resample until $\|(x, y)\|_2 > 0.17$
- `initial_state.object_velocity` = $\mathbf{0}_3$
- `initial_state.goal_position` = $[0.45, -0.05, -0.323]$
- `initial_state.goal_velocity` = $\mathbf{0}_3$
- `step_function`:
 - clips action to action-space bounds
 - applies action torques to joints
 - advances MuJoCo simulation by dt
 - updates observation vector from simulation state
 - computes reward components and total reward
 - checks for finite observation values
 - increments step count
 - returns (observation, reward, terminated=False, truncated=done_flag, info) with reward components in info
- `seeding`: via `reset(seed=...)` method only
- `info dict keys`: “reward_dist”, “reward_ctrl”, “reward_near”
- `fingertip position`: MuJoCo site “tips_arm”
- `object & goal positions`: MuJoCo bodies “object” and “goal”
- `internal state`: corresponds to MuJoCo simulation state

<Theoretical Foundations>

- **MDP tuple**: (S, A, P, R)
 - S : continuous state space \mathbb{R}^{23} (23-dimensional real vector)
 - A : continuous action space \mathbb{R}^7 with elementwise bounds $[-2, 2]$ (torques)
 - P : transition probability induced by MuJoCo physics with $dt = 0.05\text{ s}$
 - R : reward function as defined below
- **Reward function**
 - Definition:

$$r(s, a) = -\|P_{\text{object}} - P_{\text{goal}}\|_2 - 0.1\|a\|_2^2 - 0.5\|P_{\text{fingertip}} - P_{\text{object}}\|_2$$
 - Where:
 - * $P_{\text{object}}, P_{\text{goal}}, P_{\text{fingertip}} \in \mathbb{R}^3$ are positions in meters
 - * $\|\cdot\|_2$ is the Euclidean norm
 - * $a \in \mathbb{R}^7$ is the action torque vector in $N \cdot m$

- **Episode ending**

- Truncation after 100 steps or if any observation value is non-finite
- No termination condition (i.e., `terminated=False` always)
- **Fingertip position**
 - Computed via forward kinematics internally by MuJoCo
- **Symbol Table**
 - a : Action torque vector; Unit $N \cdot m$; Range: each element in $[-2, 2]$
 - $P_{\text{fingertip}}$: Fingertip 3D position; Unit m; Range: real values, unbounded
 - P_{object} : Object 3D position; Unit m; Range: real values, unbounded
 - P_{goal} : Goal 3D position; Unit m; Value: fixed at $[0.45, -0.05, -0.323]$
 - r : Reward; Unit: unitless; Range: real values (sum of negative components)
 - dt : Simulation timestep; Unit s; Value: 0.05
 - `step_count`: Current timestep count; Unit: integer; Range: 0 to 100
- **Final Specification**
 - Environment Name: `Pusher-v4`
 - Class Name: `Environment`
 - **Public API**
 - * `__init__(self)`
 - Initializes the MuJoCo simulation environment with Pusher-v4 model
 - Sets initial internal variables including step count
 - Defines action and observation spaces as specified
 - Sets fixed goal position
 - * `set_state(self, state: np.ndarray) -> None`
 - Input: state vector of shape (23,) matching observation space
 - Sets the internal MuJoCo simulation state to correspond to the given observation
 - Resets step count to zero; assumes state is valid and episode not done
 - * `step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, dict]`
 - Input: action vector of shape (7,) clipped to $[-2, 2]$
 - Applies action torques to the simulation joints and advances by $dt = 0.05$ s
 - Updates observation vector; computes reward components and total reward
 - Truncates if any observation value is non-finite or if step count ≥ 100
 - Returns: observation (23,) float64, reward float, terminated False, truncated bool, info with keys “reward_dist”, “reward_ctrl”, “reward_near”
 - **Core Logic**
 - * Simulates a 7-DOF robotic arm pushing an object to a goal
 - * State includes joint angles/velocities, fingertip, object, and goal positions
 - * Actions are joint torques clipped to $[-2, 2]$
 - * Physics advances by 0.05 s per step
 - * Reward encourages moving object to goal, penalizes large actions, and encourages fingertip proximity to object
 - * Episode truncates after 100 steps or on non-finite state
 - * Fingertip and object positions come from MuJoCo sites/bodies
 - * Seeding via `reset(seed=...)` (not shown here)
 - **Usage Scenarios & Examples**
 - * **Scenario 1: Initial step with zero action**
 - Observation: initial state vector with zero joint positions, small velocity noise, specified object and goal


```

1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565

```

```

        · Action: np.zeros(7, dtype=np.float32)
        · Reward: negative sum of distances (control penalty = 0)
        · Terminated: False; Truncated: False
        · Info: {"reward_dist": value, "reward_ctrl": 0.0,
                "reward_near": value}
    * Scenario 2: Non-zero action moves fingertip closer
        · Observation: updated state after physics step
        · Action: np.array([1.0, -0.5, 0.0, 0.2, 0.0, -1.0,
                           0.5], dtype=np.float32)
        · Reward: computed per formula
        · Terminated: False; Truncated: False (unless step count reached 100)
        · Info: {"reward_dist": value, "reward_ctrl": negative
                squared norm of action, "reward_near": value}

<Assumptions & Risks>
    · Assumes MuJoCo physics engine is used as backend with version constraints.
    · Assumes internal MuJoCo state can be set and queried to implement set_state
      and step.
    · Fingertip position is assumed to be accessible via MuJoCo site "tips_arm".
    · Reward weights are fixed as per v4; no user-configurable weights.
    · Episode termination is only by truncation; no natural termination.
    · Seeding is only via reset (seed=...); no separate seed method.
    · Risks include dependency on MuJoCo simulation correctness and numerical stability.
    · No explicit handling of invalid actions beyond clipping.
    · No rendering or visualization included.

<Library Usage>
    · Uses Python standard library and NumPy only.
    · NumPy used for array operations, clipping (np.clip), norm calculations
      (np.linalg.norm), and finite checks (np.isfinite).
    · No external dependencies beyond MuJoCo simulation backend (assumed).
    · All units are SI: torque in N·m, angles in radians, positions in meters, time in seconds.
    · No source code copied; specification based on official documentation and API standards.

</final>

```

1552 H CASE STUDY

1554 H.1 TEXT2WORLD

1555 **Task Describe.** Prepare and deliver sandwiches to a set of children: assemble sandwiches in the
 1556 kitchen (optionally gluten-free), place them on a tray, move the tray to the child's location, and
 1557 serve subject to allergy constraints. Gluten-allergic children must receive a gluten-free sandwich;
 1558 non-allergic children may receive any sandwich. Serving requires the sandwich on the tray and the
 1559 tray at the child's location; making a sandwich switches it from "notexist" to "prepared." The goal
 1560 is that all children become served.

1562 Prev. SOTA.

```

1563 (define (domain child-snack)
1564   (:requirements :typing :negative-preconditions)
1565   (:types
    child

```

```

1566     bread-portion
1567     content-portion
1568     sandwich
1569     tray
1570     place
1571 )
1572 (:constants
1573     kitchen - place
1574 )
1575 (:predicates
1576     (at_kitchen_bread ?b - bread-portion)
1577     (at_kitchen_content ?c - content-portion)
1578     (at_kitchen_sandwich ?s - sandwich)
1579     (no_gluten_bread ?b - bread-portion)
1580     (no_gluten_content ?c - content-portion)
1581     (on_tray ?s - sandwich ?t - tray)
1582     (no_gluten_sandwich ?s - sandwich)
1583     (allergic_gluten ?c - child)
1584     (not_allergic_gluten ?c - child)
1585     (served ?c - child)
1586     (waiting ?c - child ?p - place)
1587     (at ?t - tray ?p - place)
1588     (notexist ?s - sandwich)
1589 )
1590 ;; Action: make_sandwich_no_gluten
1591 (:action make_sandwich_no_gluten
1592     :parameters (?s - sandwich ?b - bread-portion ?c - content-portion)
1593     :precondition (and
1594         (at_kitchen_bread ?b)
1595         (at_kitchen_content ?c)
1596         (no_gluten_bread ?b)
1597         (no_gluten_content ?c)
1598         (notexist ?s)
1599     )
1600     :effect (and
1601         (not (at_kitchen_bread ?b))
1602         (not (at_kitchen_content ?c))
1603         (notexist ?s)
1604         (not (notexist ?s))
1605         (at_kitchen_sandwich ?s)
1606         (no_gluten_sandwich ?s)
1607     )
1608 )
1609 ;; Action: make_sandwich
1610 (:action make_sandwich
1611     :parameters (?s - sandwich ?b - bread-portion ?c - content-portion)
1612     :precondition (and
1613         (at_kitchen_bread ?b)
1614         (at_kitchen_content ?c)
1615         (notexist ?s)
1616     )
1617     :effect (and
1618         (not (at_kitchen_bread ?b))
1619         (not (at_kitchen_content ?c))
1620         (notexist ?s)
1621         (not (notexist ?s))
1622         (at_kitchen_sandwich ?s)
1623     )
1624     ;; no effect on gluten status, sandwich may or may not be
1625     ⇨ gluten-free
1626 )
1627 )

```

```

1620 ;; Action: put_on_tray
1621 (:action put_on_tray
1622   :parameters (?s - sandwich ?t - tray)
1623   :precondition (and
1624     (at_kitchen_sandwich ?s)
1625     (at ?t kitchen)
1626   )
1627   :effect (and
1628     (not (at_kitchen_sandwich ?s))
1629     (ontray ?s ?t)
1630   )
1631 )
1632 ;; Action: serve_sandwich_no_gluten
1633 (:action serve_sandwich_no_gluten
1634   :parameters (?s - sandwich ?c - child ?t - tray ?p - place)
1635   :precondition (and
1636     (ontray ?s ?t)
1637     (no_gluten_sandwich ?s)
1638     (allergic_gluten ?c)
1639     (waiting ?c ?p)
1640     (at ?t ?p)
1641   )
1642   :effect (and
1643     (not (ontray ?s ?t))
1644     (served ?c)
1645     (not (waiting ?c ?p))
1646   )
1647 )
1648 ;; Action: serve_sandwich
1649 (:action serve_sandwich
1650   :parameters (?s - sandwich ?c - child ?t - tray ?p - place)
1651   :precondition (and
1652     (ontray ?s ?t)
1653     (not (no_gluten_sandwich ?s))
1654     (not_allergic_gluten ?c)
1655     (waiting ?c ?p)
1656     (at ?t ?p)
1657   )
1658   :effect (and
1659     (not (ontray ?s ?t))
1660     (served ?c)
1661     (not (waiting ?c ?p))
1662   )
1663 )
1664 ;; Action: move_tray
1665 (:action move_tray
1666   :parameters (?t - tray ?p1 - place ?p2 - place)
1667   :precondition (at ?t ?p1)
1668   :effect (and
1669     (not (at ?t ?p1))
1670     (at ?t ?p2)
1671   )
1672 )
1673 )
1674
1675 Agent2World.
1676
1677 (define (domain child-snack)
1678   (:requirements :typing :negative-preconditions :strips)
1679   (:types
1680     child
1681     bread-portion
1682     content-portion

```

```

1674     sandwich
1675     tray
1676     place
1677 )
1678 (:constants
1679     kitchen - place
1680 )
1681 (:predicates
1682     (at_kitchen_bread ?b - bread-portion)
1683     (at_kitchen_content ?c - content-portion)
1684     (at_kitchen_sandwich ?s - sandwich)
1685     (no_gluten_bread ?b - bread-portion)
1686     (no_gluten_content ?c - content-portion)
1687     (ontray ?s - sandwich ?t - tray)
1688     (no_gluten_sandwich ?s - sandwich)
1689     (allergic_gluten ?c - child)
1690     (not_allergic_gluten ?c - child)
1691     (served ?c - child)
1692     (waiting ?c - child ?p - place)
1693     (at ?t - tray ?p - place)
1694     (notexist ?s - sandwich)
1695 )
1696 (:action make_sandwich_no_gluten
1697     :parameters (?s - sandwich ?b - bread-portion ?c - content-portion)
1698     :precondition (and
1699         (at_kitchen_bread ?b)
1700         (at_kitchen_content ?c)
1701         (no_gluten_bread ?b)
1702         (no_gluten_content ?c)
1703         (notexist ?s)
1704     )
1705     :effect (and
1706         (not (notexist ?s))
1707         (at_kitchen_sandwich ?s)
1708         (no_gluten_sandwich ?s)
1709         (not (at_kitchen_bread ?b))
1710         (not (at_kitchen_content ?c))
1711     )
1712 )
1713 (:action make_sandwich
1714     :parameters (?s - sandwich ?b - bread-portion ?c - content-portion)
1715     :precondition (and
1716         (at_kitchen_bread ?b)
1717         (at_kitchen_content ?c)
1718         (notexist ?s)
1719     )
1720     :effect (and
1721         (not (notexist ?s))
1722         (at_kitchen_sandwich ?s)
1723         (not (at_kitchen_bread ?b))
1724         (not (at_kitchen_content ?c))
1725     )
1726 )
1727 (:action put_on_tray
1728     :parameters (?s - sandwich ?t - tray)
1729     :precondition (and
1730         (at_kitchen_sandwich ?s)
1731         (at ?t kitchen)
1732     )
1733     :effect (and
1734         (ontray ?s ?t)
1735         (not (at_kitchen_sandwich ?s))
1736     )
1737 )

```

```

1728 )
1729
1730 (:action serve_sandwich_no_gluten
1731   :parameters (?s - sandwich ?c - child ?t - tray ?p - place)
1732   :precondition (and
1733     (ontray ?s ?t)
1734     (no_gluten_sandwich ?s)
1735     (allergic_gluten ?c)
1736     (waiting ?c ?p)
1737     (at ?t ?p)
1738   )
1739   :effect (and
1740     (served ?c)
1741     (not (ontray ?s ?t))
1742   )
1743 )
1744
1745 (:action serve_sandwich
1746   :parameters (?s - sandwich ?c - child ?t - tray ?p - place)
1747   :precondition (and
1748     (ontray ?s ?t)
1749     (waiting ?c ?p)
1750     (not_allergic_gluten ?c)
1751     (at ?t ?p)
1752   )
1753   :effect (and
1754     (served ?c)
1755     (not (ontray ?s ?t))
1756   )
1757 )
1758
1759 (:action move_tray
1760   :parameters (?t - tray ?p1 - place ?p2 - place)
1761   :precondition (at ?t ?p1)
1762   :effect (and
1763     (at ?t ?p2)
1764     (not (at ?t ?p1))
1765   )
1766 )
1767 )
1768 )
1769 )
1770 )
1771 )
1772 )
1773 )
1774 )
1775 )
1776 )
1777 )
1778 )
1779 )
1780 )
1781 )

```

Analysis. Compared to the baseline domain, our *Child-Snack* formulation introduces three task-aligned modifications that improve state consistency, compositionality, and plan feasibility. (i) **Creation-valid effects.** During sandwich construction we flip the existence status from “non-existent” to “prepared,” and record gluten-free status when applicable, thereby avoiding contradictory postconditions at creation time; this yields deterministic successor states and reduces backtracking caused by ill-defined truth values. (ii) **Serve-focused effects.** During serving we only transfer the item off the tray and mark the child as served, leaving the waiting label untouched; this separation of concerns prevents nonessential side-effects, preserves modular composability with downstream routines (e.g., queueing or follow-up allocation), and promotes goal-monotonic progress on the served objective. (iii) **Permissive-serving preconditions.** For non-allergic children we do not exclude gluten-free items, weakening preconditions to accept any admissible sandwich; this enlarges the feasible search space and prevents avoidable dead-ends when only gluten-free inventory remains, while safety for allergic children is still enforced via a dedicated gluten-free serving action. Collectively, these choices align with the ground-truth specification, produce cleaner state transitions, and yield empirically favorable search dynamics—smaller inconsistent-state frontiers and fewer spurious deletions—resulting in a more robust make→put-on-tray→move-tray→serve pipeline for the objective of “serving each child an acceptable sandwich.”

1778 H.2 CWMB

Task Describe. Control a 3D Ant (one free-body torso with four 2-DoF legs; nine bodies, eight hinge joints) to move forward along the +x axis by applying torques to the eight joints at each

step. The action space is $\text{Box}([-1, 1]^8)$ (joint torques). Observations list positions then velocities (27-D by default; 29-D if current x, y are included), and optionally +84 contact-force terms when `use_contact_forces=True` or `version < v4`. The reward is $r = r_{\text{healthy}} + r_{\text{forward}} - \text{ctrl_cost}$ (and $-\text{contact_cost}$ if contact forces are used), where $r_{\text{forward}} \approx \Delta x / \Delta t$ is positive for motion in $+x$ and $\Delta t = \text{frame_skip} \times 0.01 = 0.05$ by default. Episodes start from an upright, slightly noisy state, truncate at 1000 steps, and (by default) terminate early if the agent becomes unhealthy (non-finite state or torso $z \notin [0.2, 1.0]$).

Prev SOTA vs Agent2World.

```
import numpy as np
from typing import Optional, Tuple
# [D-0] Toggle: False = Ours (physics-like integration), True = Baseline
#   ↪ (simple kinematics + noise)
BASELINE = False

class Environment:
    def __init__(
        self,
        # ---- Common params ----
        frame_skip: int = 5,
        max_episode_steps: int = 1000,
        healthy_reward: float = 1.0,
        terminate_when_unhealthy: bool = True,
        healthy_z_range: Tuple[float, float] = (0.2, 1.0),
        ctrl_cost_weight: float = 0.5,
        # ---- Ours-only params ----
        contact_cost_weight: float = 5e-4,
        contact_force_range: Tuple[float, float] = (-1.0, 1.0),
        exclude_current_positions_from_observation: bool = True,
        use_contact_forces: bool = False,
        reset_noise_scale_ours: float = 0.1,
        forward_reward_weight: float = 1.0,
        # ---- Baseline-only params ----
        reset_noise_scale_baseline: float = 0.01,
        seed: Optional[int] = None,
    ):
        # Core configuration
        self.frame_skip = frame_skip
        self.dt = 0.01 * frame_skip
        self.max_episode_steps = max_episode_steps
        self.healthy_reward = healthy_reward
        self.terminate_when_unhealthy = terminate_when_unhealthy
        self.healthy_z_range = healthy_z_range
        self.ctrl_cost_weight = ctrl_cost_weight
        self.forward_reward_weight = forward_reward_weight # [D-5] Only
        #   ↪ used by Ours.
        self.np_random = np.random.RandomState(seed)

        # [D-1] Observation schema differs:
        #   Baseline: fixed 27 = 13 positions (z, quat4, joint8) + 14
        #   ↪ velocities (flat vector).
        #   Ours: positions(15) + velocities(14) (+ optional
        #   ↪ torso_xy, contact forces),
        #   with option to exclude torso x,y from observation.
        self.exclude_current_positions_from_observation = (
            False if BASELINE else
            #   ↪ bool(exclude_current_positions_from_observation)
        )

        # [D-2] Contacts: Baseline has no contact forces/cost; Ours can
        #   ↪ include 84-dim contact forces + cost.
        self.use_contact_forces = False if BASELINE else
        #   ↪ bool(use_contact_forces)
        self.contact_cost_weight = 0.0 if BASELINE else
        #   ↪ float(contact_cost_weight)
```

```

1836     self.contact_force_range = contact_force_range
1837
1838     self.step_count = 0
1839
1840     if BASELINE:
1841         # Baseline state: flat (27,) observation vector
1842         self.obs_shape = (27,)
1843         self.state = np.zeros(self.obs_shape, dtype=np.float64)
1844         self.x_position = 0.0 # [D-4] Progress tracked separately
1845         ↪ (not in observation)
1846         self.y_position = 0.0
1847         self.reset_noise_scale = float(reset_noise_scale_baseline) #
1848         ↪ [D-4]
1849         self.contact_forces = None
1850         self.observation_dim = 27 # [D-1]
1851     else:
1852         # Ours state: split positions(15) / velocities(14)
1853         self.pos_dim = 15 # torso_pos(3), torso_quat(4),
1854         ↪ joint_angles(8)
1855         self.vel_dim = 14 # torso_lin_vel(3), torso_ang_vel(3),
1856         ↪ joint_vel(8)
1857         self.positions = np.zeros(self.pos_dim, dtype=np.float64)
1858         self.velocities = np.zeros(self.vel_dim, dtype=np.float64)
1859         self.reset_noise_scale = float(reset_noise_scale_ours) #
1860         ↪ [D-4]
1861         self.last_x_position = 0.0 # [D-4] (used when restoring
1862         ↪ state)
1863         self.contact_forces = (
1864             np.zeros(84, dtype=np.float64) if self.use_contact_forces
1865             ↪ else None
1866         )
1867         # Compute observation length for Ours
1868         base_pos_len = self.pos_dim
1869         if self.exclude_current_positions_from_observation:
1870             base_pos_len -= 2 # drop torso x,y
1871         self.obs_pos_len = base_pos_len
1872         self.obs_vel_len = self.vel_dim
1873         self.obs_contact_len = 84 if self.use_contact_forces else 0
1874         self.obs_torso_xy_len = 0 if
1875         ↪ self.exclude_current_positions_from_observation else 2
1876         self.observation_dim = (
1877             self.obs_pos_len + self.obs_vel_len +
1878             ↪ self.obs_torso_xy_len + self.obs_contact_len
1879         )
1880
1881     self.reset(seed)
1882
1883     # ----- Helpers (shared) -----
1884     def _is_healthy(self, z_value: Optional[float] = None) -> bool:
1885         # [D-3] Health z-source differs:
1886         # Baseline: use state[0] (z) from flat observation.
1887         # Ours: use positions[2] (torso z) from split state.
1888         if BASELINE:
1889             st = self.state
1890             if not np.all(np.isfinite(st)):
1891                 return False
1892             z = st[0] if z_value is None else z_value
1893             return self.healthy_z_range[0] <= z <=
1894                 ↪ self.healthy_z_range[1]
1895         else:
1896             if not np.all(np.isfinite(self.positions)) or not
1897             ↪ np.all(np.isfinite(self.velocities)):
1898                 return False
1899             z = self.positions[2]
1900             return self.healthy_z_range[0] <= z <=
1901                 ↪ self.healthy_z_range[1]

```

```

1890
1891 def _get_observation(self) -> np.ndarray:
1892     # [D-1] Observation layout differs (see __init__ comment).
1893     if BASELINE:
1894         return self.state.copy()
1895     if self.exclude_current_positions_from_observation:
1896         pos_obs = self.positions[2:].copy() # exclude torso x,y
1897         torso_xy = np.array([], dtype=np.float64)
1898     else:
1899         pos_obs = self.positions.copy()
1900         torso_xy = self.positions[0:2].copy()
1901     vel_obs = self.velocities.copy()
1902     obs = np.concatenate((pos_obs, vel_obs))
1903     if not self.exclude_current_positions_from_observation:
1904         obs = np.concatenate((obs, torso_xy))
1905     if self.use_contact_forces:
1906         obs = np.concatenate((obs, self.contact_forces))
1907     return obs
1908
1909 @staticmethod
1910 def _quat_multiply(q1: np.ndarray, q2: np.ndarray) -> np.ndarray:
1911     w1, x1, y1, z1 = q1
1912     w2, x2, y2, z2 = q2
1913     return np.array(
1914         [
1915             w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2,
1916             w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2,
1917             w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2,
1918             w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2,
1919         ],
1920         dtype=np.float64,
1921     )
1922
1923 # ----- Public API -----
1924 def set_state(self, state: np.ndarray) -> None:
1925     state = np.asarray(state, dtype=np.float64)
1926     if BASELINE:
1927         # [D-8] Baseline expects a flat 27-dim observation
1928         ↪ (pos13+vel14).
1929         if state.shape != (27,):
1930             raise ValueError(f"set_state input must have shape (27,),
1931                               ↪ got {state.shape}")
1932         if not np.all(np.isfinite(state)):
1933             raise ValueError("set_state input contains non-finite
1934                               ↪ values")
1935         self.state = state.copy()
1936         self.step_count = 0
1937         self.x_position = 0.0 # [D-4] progress variable is external
1938         ↪ to obs
1939         self.y_position = 0.0
1940     else:
1941         # [D-8] Ours expects the current obs layout length and
1942         ↪ reconstructs split state.
1943         expected_len = self.observation_dim
1944         if state.ndim != 1 or state.shape[0] != expected_len:
1945             raise ValueError(f"State must be 1D array of length
1946                               ↪ {expected_len}, got shape {state.shape}")
1947         pos_len = self.obs_pos_len
1948         vel_len = self.obs_vel_len
1949         pos_part = state[:pos_len]
1950         vel_part = state[pos_len : pos_len + vel_len]
1951         if self.exclude_current_positions_from_observation:
1952             full_positions = np.zeros(self.pos_dim, dtype=np.float64)
1953             full_positions[2:] = pos_part
1954             full_positions[0] = 0.0
1955             full_positions[1] = 0.0

```



```

1944         else:
1945             full_positions = pos_part.copy()
1946             self.positions = full_positions
1947             self.velocities = vel_part.copy()
1948             self.step_count = 0
1949             self.last_x_position = self.positions[0] # [D-4]
1950
1951 def reset(self, seed: Optional[int] = None) -> np.ndarray:
1952     if seed is not None:
1953         self.np_random.seed(seed)
1954     self.step_count = 0
1955
1956     if BASELINE:
1957         # [D-2][D-4] Baseline init: 13 pos (z, quat, joints) + 14
1958         ↪ vel; small noise.
1959         pos = np.zeros(13, dtype=np.float64)
1960         pos[0] = 0.75 # z
1961         pos[1:5] = np.array([1.0, 0.0, 0.0, 0.0]) # quaternion
1962         ↪ (w,x,y,z)
1963         noise_pos = self.np_random.uniform(-self.reset_noise_scale,
1964             ↪ self.reset_noise_scale, size=13)
1965         pos = pos + noise_pos
1966         vel = self.np_random.normal(0, self.reset_noise_scale,
1967             ↪ size=14)
1968         self.state = np.concatenate([pos, vel])
1969         self.x_position = 0.0 # [D-4] progress variable
1970         self.y_position = 0.0
1971     else:
1972         # Ours init: positions(15) / velocities(14) with larger noise
1973         ↪ and full torso pose.
1974         base_positions = np.zeros(15, dtype=np.float64)
1975         base_positions[2] = 0.75 # torso z
1976         base_positions[3] = 1.0 # quat.w
1977         noise_pos = self.np_random.uniform(-self.reset_noise_scale,
1978             ↪ self.reset_noise_scale, size=15)
1979         self.positions = base_positions + noise_pos
1980         self.velocities = self.np_random.normal(loc=0.0,
1981             ↪ scale=self.reset_noise_scale, size=14)
1982         self.last_x_position = self.positions[0] # [D-4]
1983         if self.use_contact_forces and self.contact_forces is not
1984             ↪ None:
1985             self.contact_forces[:] = 0.0
1986     return self._get_observation()
1987
1988 def step(self, action: np.ndarray):
1989     action = np.asarray(action, dtype=np.float64)
1990     if action.shape != (8,):
1991         raise ValueError(f"Action must be of shape (8,), got
1992             ↪ {action.shape}")
1993
1994     # [D-6] Action bound handling differs:
1995     # Baseline: out-of-bounds raises; Ours: clip to [-1, 1].
1996     if BASELINE:
1997         if np.any(action < -1.0) or np.any(action > 1.0):
1998             raise ValueError("Action values must be in [-1, 1] for
1999                 ↪ Baseline")
2000     else:
2001         action = np.clip(action, -1.0, 1.0)
2002
2003     # ---- Inline divergence (single function, two branches) ----
2004     if BASELINE:
2005         # Forward progress proxy from hip joints
2006         prev_x_pos = self.x_position # [D-4] external tracker (not
2007             ↪ in obs)
2008         forward_force = float(np.sum(action[[0, 2, 4, 6]]))

```

```

1998         self.x_position += forward_force * self.dt * 0.1 # arbitrary
1999         ↪ scale
2000
2001         # Stochastic updates (no true physics)
2002         z = float(np.clip(self.state[0] +
2003         ↪ self.np_random.uniform(-0.01, 0.01), 0.0, 2.0))
2004         # [D-7] Orientation update: Baseline = add noise to
2005         ↪ quaternion then renormalize.
2006         orientation = self.state[1:5] + self.np_random.uniform(-0.01,
2007         ↪ 0.01, size=4)
2008         norm = float(np.linalg.norm(orientation))
2009         orientation = orientation / norm if norm > 0 else
2010         ↪ np.array([1.0, 0.0, 0.0, 0.0], dtype=np.float64)
2011         # Joint angles: integrate action + small noise; wrap to [-pi,
2012         ↪ pi]
2013         joint_angles = self.state[5:13] + action * self.dt +
2014         ↪ self.np_random.uniform(-0.005, 0.005, size=8)
2015         joint_angles = (joint_angles + np.pi) % (2 * np.pi) - np.pi
2016         # Velocities: torso (noise) + joints (action + noise)
2017         torso_vel = self.np_random.normal(0, 0.01, size=6)
2018         joint_vel = action + self.np_random.normal(0, 0.01, size=8)
2019         velocities = np.concatenate([torso_vel, joint_vel])
2020         # Compose new flat state
2021         pos = np.empty(13, dtype=np.float64)
2022         pos[0] = z
2023         pos[1:5] = orientation
2024         pos[5:13] = joint_angles
2025         self.state = np.concatenate([pos, velocities])
2026
2027         healthy = self._is_healthy(z_value=z) # [D-3]
2028         forward_delta = self.x_position - prev_x_pos
2029         contact_cost = 0.0 # [D-2] no contacts in Baseline
2030         weight = 1.0 # [D-5] forward reward weight fixed to
2031         ↪ 1.0
2032     else:
2033         # Physics-like integration
2034         old_x = float(self.positions[0]) # [D-4] directly from torso
2035         ↪ x in positions
2036         # Joint dynamics: dv = (u - damp*v)*dt; dq = v*dt
2037         joint_damping = 0.1
2038         joint_vel_prev = self.velocities[6:]
2039         joint_acc = action - joint_damping * joint_vel_prev # [D-1]
2040         ↪ acts on split state
2041         joint_vel_new = joint_vel_prev + self.dt * joint_acc
2042         joint_ang_new = self.positions[7:] + self.dt * joint_vel_new
2043         # Torso linear & angular velocity damping
2044         lin_damping = 0.1
2045         ang_damping = 0.1
2046         torso_lin_vel_new = self.velocities[0:3] * (1 - lin_damping *
2047         ↪ self.dt)
2048         torso_ang_vel_new = self.velocities[3:6] * (1 - ang_damping *
2049         ↪ self.dt)
2050         # Integrate torso position
2051         torso_pos_new = self.positions[0:3] + self.dt *
2052         ↪ torso_lin_vel_new
2053         # [D-7] Orientation update: Ours = quaternion integration
2054         ↪ from angular velocity.
2055         q = self.positions[3:7]
2056         omega = torso_ang_vel_new
2057         omega_quat = np.array([0.0, omega[0], omega[1], omega[2]],
2058         ↪ dtype=np.float64)
2059         q_dot = 0.5 * self._quat_multiply(omega_quat, q)
2060         q_new = q + self.dt * q_dot
2061         norm = float(np.linalg.norm(q_new))
2062         q_new = q_new / norm if norm > 0 else np.array([1.0, 0.0,
2063         ↪ 0.0, 0.0], dtype=np.float64)

```

```

2052     # Write back split state
2053     self.positions[0:3] = torso_pos_new
2054     self.positions[3:7] = q_new
2055     self.positions[7:] = joint_ang_new
2056     self.velocities[0:3] = torso_lin_vel_new
2057     self.velocities[3:6] = torso_ang_vel_new
2058     self.velocities[6:] = joint_vel_new
2059     # Contacts (optional)
2060     if self.use_contact_forces and self.contact_forces is not
2061         ↪ None:
2062         self.contact_forces.fill(0.0) # [D-2]
2063     new_x = float(self.positions[0])
2064     healthy = self._is_healthy() # [D-3]
2065     forward_delta = new_x - old_x
2066     # Contact cost (if enabled)
2067     if self.use_contact_forces and self.contact_forces is not
2068         ↪ None:
2069         clipped = np.clip(self.contact_forces,
2070             ↪ self.contact_force_range[0],
2071             ↪ self.contact_force_range[1])
2072         contact_cost = self.contact_cost_weight *
2073             ↪ float(np.sum(np.square(clipped))) # [D-2]
2074     else:
2075         contact_cost = 0.0
2076     weight = self.forward_reward_weight # [D-5]
2077
2078     # ---- Shared reward & termination (single exit) ----
2079     forward_reward = weight * (forward_delta / self.dt) # [D-5]
2080     ctrl_cost = self.ctrl_cost_weight *
2081         ↪ float(np.sum(np.square(action)))
2082     reward = (self.healthy_reward if healthy else 0.0) +
2083         ↪ forward_reward - ctrl_cost - contact_cost
2084
2085     self.step_count += 1
2086     done = (self.terminate_when_unhealthy and not healthy) or
2087         ↪ (self.step_count >= self.max_episode_steps)
2088     return self._get_observation(), reward, done

```

Analysis. On the **Ant-v4** forward-locomotion task, AGENT2WORLD surpasses the *Baseline* with higher success, smoother gait, and lower energy per meter under identical horizons and z -health checks. (i) **State & sensing.** The *Baseline* exposes a flat 27-D observation, while we adopt a task-aligned layout that separates positions/velocities and can hide global (x, y) by default ([D-1]). We additionally support contact forces for foot-ground cues ([D-2]). Health uses torso z from split state rather than the flat vector slot ([D-3]). State restoration matches each layout: the *Baseline* ingests a 27-D vector, whereas ours reconstructs split buffers from the current observation setting ([D-8]). (ii) **Dynamics & orientation.** The *Baseline* updates orientation by quaternion noise plus renormalization, and treats actions as noisy joint velocities; we integrate damped joint accelerations and update attitude via $\dot{\mathbf{q}} = \frac{1}{2} \boldsymbol{\omega}_q \otimes \mathbf{q}$ with renormalization ([D-7]). This physically consistent pipeline—enabled by the split state design ([D-1])—low-passes high-frequency actuation, reduces roll/pitch jitter, and yields more phase-coordinated gaits. (iii) **Control semantics & reward.** The *Baseline* hard-errors on out-of-range actions and uses a fixed forward-reward weight; forward progress is tracked by an external x variable and reset noise is smaller. Ours clips actions to $[-1, 1]$ ([D-6]), uses a tunable forward-reward weight ([D-5]), measures progress directly from torso x in the state and employs a different reset scale ([D-4]); an optional contact-cost term can be included when contact signals are enabled ([D-2]). Together these choices stabilize training signals and improve sample efficiency.

Summary of diffs. [D-1] Observation schema: Baseline uses a flat 27-D vector; Ours uses split positions+velocities with optional hidden (x, y) and optional contact forces; [D-2] Contacts: Baseline has no contact forces/cost; Ours optionally exposes 84-D contact forces and a contact-cost term; [D-3] Health source: Baseline takes z from the flat vector slot; Ours uses torso z from split positions; [D-4] Progress & reset: Baseline tracks forward x as an external variable and uses smaller reset noise; Ours reads torso x from state and uses a different reset scale; [D-5] Forward-reward

weight: Baseline fixed to 1.0; Ours is tunable; [D-6] Action bounds: Baseline errors on out-of-range actions; Ours clips to $[-1, 1]$; [D-7] Orientation update: Baseline adds noise then renormalizes quaternion; Ours integrates $\dot{\mathbf{q}} = \frac{1}{2} \omega_q \otimes \mathbf{q}$ then renormalizes; [D-8] State setting: Baseline ingests a flat 27-D state; Ours reconstructs split buffers from the current observation layout.

H.3 BYTESIZED32

Task Description. We build a lightweight, text-interactive micro-simulation of pea growth in a small garden. The world contains a Pea, a FlowerPot, a Jug, and a Sink; water is represented as scalar levels in the Jug and FlowerPot and as an internal level in the Pea. The agent can look/examine, take/put objects, switch the sink on/off, fill the jug from the sink (effective only when the sink is on), and pour water from the jug into the flower pot. After each action, a tick advances processes: the sink supplies water if on; the pot passively transfers its water to the pea; and the pea consumes water and progresses from seed \rightarrow sprout \rightarrow young plant \rightarrow mature \rightarrow reproducing when sufficiently hydrated for several consecutive ticks. Episodes start with an unplanted pea and an empty pot; the goal is to plant the pea and water it repeatedly until it reaches the reproducing stage.

Prev SOTA vs Agent2World.

```
import random

# [D0] Toggle: False = Ours, True = Baseline
BASELINE = False

# ----- Core object model (minimal API) -----
class GameObject:
    def __init__(self, name):
        self.name, self.parent, self.contains = name, None, []
        self.props = {"isContainer": False, "isMoveable": True}
    def get(self, k, d=None):
        return self.props.get(k, d)
    def add(self, obj):
        obj.removeSelf(); self.contains.append(obj);
        obj.parent = self
    def remove(self, obj):
        self.contains.remove(obj); obj.parent = None
    def removeSelf(self):
        if self.parent: self.parent.remove(self)
    def allContained(self):
        out = []
        for o in self.contains: out += [o] + o.allContained()
        return out
    def tick(self): pass

class Container(GameObject):
    def __init__(self, name): super().__init__(name);
        self.props["isContainer"] = True
    def place(self, obj):
        if not obj.get("isMoveable"):
            return ("Can't move that object.", False)
        self.add(obj); return ("OK.", True)
    def take(self, obj):
        if obj not in self.contains:
            return ("Object not here.", None, False)
        if not obj.get("isMoveable"):
            return ("Can't move that object.", None, False)
        obj.removeSelf(); return ("OK.", obj, True)

class Device(Container):
    def __init__(self, name): super().__init__(name);
        self.props.update({"isDevice": True, "isOn": False})
    def turnOn(self):
        if self.props["isOn"]:
            return (f"{self.name} is already on.", False)
        self.props["isOn"] = True
```

```

2160         return (f"{self.name} turned on.", True)
2161     def turnOff(self):
2162         if not self.props["isOn"]:
2163             return (f"{self.name} is already off.", False)
2164         self.props["isOn"] = False
2165         return (f"{self.name} turned off.", True)
2166
2167     class World(Container):
2168         def __init__(self): super().__init__("world")
2169
2170     class Agent(Container):
2171         def __init__(self): super().__init__("entity")
2172
2173     # ----- Task objects -----
2174     class Pea(GameObject):
2175         STAGES = ["seed", "sprout", "young plant", "mature plant", "reproducing"]
2176         MAX_WATER, CONSUME, NEED, TICKS = 100, 5, 30, 3
2177         def __init__(self):
2178             super().__init__("pea"); self.props["isMoveable"] = True
2179             self.stage, self.water, self.hydrated = 0, 0, 0
2180         @property
2181         def stage_name(self):
2182             return self.STAGES[self.stage]
2183         def addWater(self, n):
2184             self.water = min(self.water + n, self.MAX_WATER)
2185         def tick(self):
2186             # [D3] Growth rule: Baseline = simple ( $\geq 2 \rightarrow +\text{stage}$ , else  $-1$  if  $\hookrightarrow > 0$ ); Ours = threshold + accumulation.
2187             if BASELINE:
2188                 if self.stage < len(self.STAGES)-1:
2189                     if self.water  $\geq$  2: self.water -= 2; self.stage += 1
2190                     elif self.water > 0: self.water -= 1
2191                 return
2192             self.water = max(self.water - self.CONSUME, 0)
2193             if self.water  $\geq$  self.NEED:
2194                 self.hydrated += 1
2195                 if self.hydrated  $\geq$  self.TICKS and self.stage < len(self.STAGES)-1:
2196                     self.stage += 1; self.hydrated = 0
2197
2198     class FlowerPot(Container):
2199         MAX_WATER = 100
2200         def __init__(self):
2201             super().__init__("flower pot")
2202             self.water = 0
2203         def addWater(self, n):
2204             add = min(self.MAX_WATER - self.water, n);
2205             self.water += add; return add
2206         def consume(self, n):
2207             use = min(self.water, n); self.water -= use; return use
2208         def tick(self):
2209             # [D2] Passive transfer: Baseline = none; Ours = transfer  $\hookrightarrow$  pot.water to pea on each tick.
2210             if BASELINE: return
2211             pea = next((o for o in self.contains if isinstance(o, Pea)),
2212                        $\hookrightarrow$  None)
2213             if pea and self.water > 0:
2214                 x = self.consume(min(self.water, Pea.MAX_WATER))
2215                 pea.addWater(x)
2216
2217     class Jug(Container):
2218         MAX_WATER = 100
2219         def __init__(self):
2220             super().__init__("jug")
2221             self.water = 0
2222         def fill(self, n):

```

```

2214         add = min(self.MAX_WATER - self.water, n)
2215         self.water += add
2216         return add
2217     def pour(self, n):
2218         out = min(self.water, n)
2219         self.water -= out
2220         return out
2221
2221 class Sink(Device):
2222     MAX_WATER = 1000
2223     def __init__(self): super().__init__("sink");
2224     self.water = self.MAX_WATER
2225     def tick(self):
2226         self.water = self.MAX_WATER if self.props["isOn"] else 0
2227
2227 # ----- Minimal game scaffold (only actions we need) -----
2228 class TextGame:
2229     MAX_STEPS = 50
2230     def __init__(self, seed=0):
2231         random.seed(seed)
2232         self.world, self.agent = World(), Agent();
2233         ↪ self.world.add(self.agent)
2234         self.pea, self.pot, self.jug, self.sink = Pea(), FlowerPot(),
2235         ↪ Jug(), Sink()
2236         # [D7] Movability: Baseline pins the sink as immovable (ours
2237         ↪ keeps defaults).
2238         if BASELINE: self.sink.props["isMoveable"] = False
2239         for o in (self.pot, self.jug, self.sink, self.pea):
2240             self.world.add(o)
2241         self.score = self.steps = 0
2242         self.over = self.won = False
2243
2244 # API of interest (matching both variants); unchanged helpers omitted
2245 ↪ for brevity.
2246 def _obj(self, name):
2247     for o in [self.world] + self.world.allContained():
2248         if o.name == name: return o
2249     for o in self.agent.contains:
2250         if o.name == name: return o
2251     return None
2252
2253 def calculateScore(self):
2254     # [D5] Reward: Baseline = stage*10; Ours = stage*20 + water bonus
2255     ↪ (<=20).
2256     if BASELINE:
2257         self.score = self.pea.stage*10
2258     else:
2259         self.score = self.pea.stage*20 +
2260         ↪ int(self.pea.water/Pea.MAX_WATER*20)
2261     if self.pea.stage_name == "reproducing":
2262         self.won = self.over = True
2263     if self.steps >= self.MAX_STEPS and not self.won:
2264         self.over = True
2265
2266 # ----- actions -----
2267 def take(self, name):
2268     o = self._obj(name);
2269     if not o: return f"No {name}."
2270     if not o.get("isMoveable"): return f"Can't take {name}."
2271     if o.parent != self.world: return f"{name} not here."
2272     _, got, ok = self.world.take(o);
2273     if ok: self.agent.add(got);
2274     return "OK." if ok else "Fail."
2275
2276 def put(self, obj, cont):
2277     o, c = self._obj(obj), self._obj(cont)

```

```

2268         if not o or o.parent != self.agent:
2269             return f"No {obj} in inventory."
2270         if not c or not c.get("isContainer"):
2271             return f"{cont} not a container."
2272         # [D6] Placement constraint: Ours restricts pea -> pot only;
2273         ↪ Baseline has no special rule.
2274         if (not BASELINE) and isinstance(o, Pea) and not isinstance(c,
2275             ↪ FlowerPot):
2276             return "Pea must go into flower pot."
2277         _, ok = c.place(o); return "OK." if ok else "Fail."
2278
2279     def turn_on(self, dev):
2280         d = self._obj(dev);
2281         if not d or not d.get("isDevice"): return f"No device {dev}."
2282         msg,_ = d.turnOn(); return msg
2283     def turn_off(self, dev):
2284         d = self._obj(dev);
2285         if not d or not d.get("isDevice"): return f"No device {dev}."
2286         msg,_ = d.turnOff(); return msg
2287
2288     def fill_from_sink(self):
2289         # [D1] Fill gating: Baseline ignores sink.on; Ours requires
2290         ↪ sink.on == True.
2291         if (not BASELINE) and (not self.sink.props["isOn"]): return "Sink
2292         ↪ is off."
2293         need = self.jug.MAX_WATER - self.jug.water
2294         if need <= 0: return "Jug already full."
2295         self.jug.fill(need) # treat sink as infinite when allowed
2296         return "Jug filled."
2297
2298     def pour_to_pot(self):
2299         if self.jug.water <= 0: return "Jug empty."
2300         # [D8] Pour semantics: Baseline feeds pea directly; Ours fills
2301         ↪ pot; pea drinks via [D2].
2302         poured = self.jug.pour(10)
2303         if BASELINE and (self.pea in self.pot.contains):
2304             self.pea.addWater(3); return "Poured; pea absorbs water."
2305         added = self.pot.addWater(poured)
2306         if added < poured: self.jug.fill(poured - added)
2307         return "Poured into pot."
2308
2309     # ----- driver -----
2310     def step(self, cmd):
2311         self.steps += 1
2312         # [D4] Update order: Baseline ticks BEFORE action; Ours ticks
2313         ↪ AFTER.
2314         if BASELINE:
2315             for o in [self.world] + self.world.allContained(): o.tick()
2316
2317         parts = cmd.lower().strip().split()
2318         out = "Unknown."
2319         try:
2320             if parts[:1]==["take"]: out = self.take(" ".join(parts[1:]))
2321             elif parts[:1]==["put"] and "in" in parts:
2322                 i = parts.index("in");
2323                 out = self.put(" ".join(parts[1:i]), "
2324                 ↪ ".join(parts[i+1:]))
2325             elif parts[:2]==["turn","on"]:
2326                 out = self.turn_on(" ".join(parts[2:]))
2327             elif parts[:2]==["turn","off"]:
2328                 out = self.turn_off(" ".join(parts[2:]))
2329             elif parts[:3]==["fill","jug","from"]:
2330                 out = self.fill_from_sink()
2331             elif parts[:4]==["pour","water","from","jug"] and "in" in
2332                 ↪ parts:
2333                 out = self.pour_to_pot()

```

```

2322         # distractor (spec only)
2323         elif parts[:1]=="use": out = "Nothing happens."
2324     except Exception as e:
2325         out = f"Error: {e}"
2326
2327     if not BASELINE:
2328         for o in [self.world] + self.world.allContained(): o.tick()
2329         old = self.score; self.calculateScore();
2330         reward = self.score - old
2331         return out, self.score, reward, self.over, self.won

```

Analysis. Under identical initialization and evaluation (capacity limits and preconditions enforced), AGENT2WORLD outperforms a *Baseline* on the pea-growing (water-transfer) task, yielding higher success, shorter trajectories, and fewer invalid actions. (i) **Action space and dynamics.** We expose a precondition-aware interface and decouple water flow from uptake: `fill` is effective only when the sink is on ([D1]); `pour` increases the pot’s water and the pea hydrates asynchronously via `tick` ([D2], [D8]). We advance environment dynamics *after* the action to preserve causal credit assignment ([D4]). By contrast, the *Baseline* exposes `fill` irrespective of sink state, credits hydration at pour time, and updates *before* acting. (ii) **Physical consistency and constraints.** We enforce finite capacities with overflow returned to the jug and constrain placement so the pea can only be planted in the flower pot ([D6]). These constraints prune degenerate branches without removing valid solutions. The *Baseline* omits the planting constraint and hydrates synchronously, which increases misleading transitions. (Regarding movability, the *Baseline* pins the sink as immovable while Ours keeps defaults; this ablation affects search but not preconditions, [D7]). (iii) **Growth model and reward.** Plant physiology follows thresholded, accumulated growth with per-tick water consumption ([D3]). Reward shaping combines stage progress with a bounded water bonus, and immediate rewards are score deltas ([D5]). The *Baseline* uses a stage-only score without water shaping, weakening the learning signal.

Summary of diffs: [D1] preconditioned `fill`; [D2] passive pot→pea transfer; [D3] threshold+consumption growth; [D4] post-action ticking; [D5] shaped reward (stage+water); [D6] pea→pot placement constraint; [D7] sink movability ablation; [D8] `pour` affects pot first (not the pea).