OBS2RULE: BENCHMARKING THE RULE DISCOVERY ABILITY OF LLM AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

For agents aiming to conduct scientific discovery autonomously, uncovering a new law requires designing experiments, seeing through complex experimental observations, and finally extracting the regularity that produced them. We formalize this process as **rule discovery**, comprising two stages: **experiment design**, deciding which observations to collect and how to collect them systematically, and **rule induction**, compressing the accumulated evidence into a general statement that also holds for cases never observed. To evaluate both stages jointly, we propose `Obs2Rule`, a text-based simulation benchmark of 32 manually designed environments spanning 12 scenarios. Each environment offers no goal to achieve and no reward; the agent must autonomously design and carry out experiments to uncover the latent rule that governs the environment, and express the induced rule as executable Python code, which is then verified against manually crafted test cases. This design decouples rule discovery from the task completion measured by most existing scientific discovery benchmarks, and provides an objective, generalization-sensitive criterion for grading induced rules. Evaluating four LLM-driven agents, we find that the best agent passes only 66% of environments, and just 53% of those requiring active intervention. Further analysis shows that current agents can articulate rules once given adequate evidence, but are weak at designing and executing the systematic experiments needed to obtain it.

1 INTRODUCTION

Scientific discovery is not merely about solving problems; it is about deciding which experiments will reveal how the world works, and turning their outcomes into a law. Large language model (LLM) agents have recently been deployed across substantial portions of the research pipeline, from literature synthesis and hypothesis generation to experimental planning and laboratory control (Wang et al., 2024; Liu et al., 2025; Gottweis et al., 2026; Lu et al., 2026), and they are increasingly evaluated on their ability to reach correct answers in scientific settings. Whether they can perform the abstraction step itself, however, remains largely untested.

Existing benchmarks for scientific and interactive reasoning such as ScienceWorld (Wang et al., 2022) adopt task completion as their measure of success: an agent is placed in an environment or handed a problem and is scored on whether it reaches a goal state, produces the right answer, or maximizes a reward. Such measures are convenient, but they do not reveal whether an agent can discover new rules itself. An agent may solve a task by recalling a solution encountered during pretraining, by exploiting a shortcut that happens to work in the particular configuration presented, or by trial and error that never consolidates into any statement about how the world behaves. Conversely, an agent that has correctly grasped the governing law may still fail for reasons of planning or execution. Task success is therefore neither necessary nor sufficient evidence that the underlying regularity has been understood, and benchmarks built on it cannot tell us whether an agent is capable of the inductive leap that discovery demands.

We argue instead for evaluating rule discovery directly, and we formalize it as a two-stage process. The first stage is experiment design: the agent must decide which observations are worth collecting and how to collect them systematically, designing experiments that isolate individual factors and cover the space of relevant conditions. The second is rule induction: compressing the accumulated evidence into a general and precise statement that also holds for cases never observed. This decom-

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.

position mirrors the classical view of discovery as coordinated search in an experiment space and a hypothesis space (Klahr & Dunbar, 1988). Existing work addresses the two stages in isolation. Symbolic regression and program-synthesized world models ask a system to produce an explicit rule, but supply the observations as a fixed dataset or as trajectories collected incidentally while pursuing a task (Udrescu & Tegmark, 2020; Tsividis et al., 2021; Tang et al., 2024; Shojaee et al., 2025); studies of exploration and active learning examine how data is acquired, but never require the agent to articulate what it has learned (Krishnamurthy et al., 2024; Roohani et al., 2025). Yet the two stages are inseparable in practice: a flawless inductive step applied to biased or incomplete evidence yields a confidently wrong rule, and in real science a conclusion is only as trustworthy as the experiments behind it.

Evaluating the two stages jointly raises the question of how an induced rule should be graded. Natural language descriptions are not well-suited to this purpose: they may be vague or only partially correct, and judging them is inherently subjective. Executable code, by contrast, is unambiguous, can be applied to inputs the agent never observed, and can be checked automatically against ground truth, making it a criterion sensitive to generality as well as correctness.

Following these principles, we introduce `Obs2Rule`, a text-based simulation benchmark of 32 manually designed environments spanning 12 scenarios. Each environment is best understood not as a game to be won but as a laboratory to be used. The agent is placed in an initially empty grid world in which it may freely create and remove objects, set up whatever configurations it deems informative, and observe the consequences of its interventions. Moreover, there is no in-environment objective and no reward: the environment contains nothing to accomplish. The agent's sole task is external to the simulation—to determine the latent rule governing some aspect of the world and to submit that rule as a Python function, which is then verified against test cases written by human annotators who enumerated the relevant combinations of conditions. Because the agent must decide for itself which experiments to run, `Obs2Rule` evaluates experiment design and rule induction jointly, and because each scenario appears in several variations that look identical but obey different rules, including one that deliberately contradicts common sense, success of which requires empirical grounding rather than recall of prior knowledge.

Our experiments with four LLM-driven agents localize the failure squarely in the first stage. Agents rarely design protocols that systematically cover the space of relevant conditions, generalize readily from a handful of incidental observations, and leave entire branches of a rule's condition space unprobed. When we instead supply human-collected observations covering every corner case and ask only for the rule, pass rates on scenarios requiring active intervention rise from 0.33 to 0.85, and the ranking of models reverses: the agents weakest at discovery as a whole are among the strongest at induction alone. For today's agents, the bottleneck of rule discovery is not expressing a rule given adequate evidence, but the disciplined collection of the evidence needed to identify it.

Our contributions are as follows.

- We formulate rule discovery as a first-class evaluation target, distinct from task completion, decompose it into experiment design and rule induction, and propose code-based rule submission as an objective, generalization-sensitive criterion for grading induced rules.
- We release `Obs2Rule`, a benchmark of 32 text-based environments across 12 scenarios covering active, passive, and stochastic modes of inquiry, together with human-authored test suites that verify submitted rules on exhaustively enumerated conditions. Counter-intuitive variations ensure that success requires empirical grounding rather than prior knowledge.
- We benchmark four LLM-driven agents and show that their principal limitation is systematic experiment design rather than rule induction, identifying a target for future work on autonomous scientific agents.

2 Related Work

Benchmarks for LLM-driven scientific discovery. A growing body of work evaluates LLM agents as scientists. ScienceWorld (Wang et al., 2022) embedsscience curricula in interactive environments. MLAgentBench (Huang et al., 2024), ScienceAgentBench (Chen et al., 2025) focus on machine learning or data-driven scientific discovery. In all of these benchmarks, success is defined by an

outcome, such as reaching a goal state or improving a metric on a dataset. No explicit rule is ever produced, and an agent can succeed with only a partial understanding of the environment. Symbolic regression benchmarks such as SRBench (La Cava et al., 2021) and LLM-SRBench (Shojaee et al., 2025) come closer by asking for a law as the output, but the observations are supplied as a fixed dataset, so the agent never decides which measurements to take. DiscoveryWorld (Jansen et al., 2024) moved one step further: alongside task completion, it scores agents on a rubric of binary explanatory questions judged by an LLM, thereby probing whether the agent understood the mechanism rather than merely exploited it. Such evaluation process is costly and its reliability rests on the judge model's own capabilities. `Obs2Rule` instead requires agents to write rules down as executable code and grades it against objective test cases.

Text games as interactive LLM benchmarks. Text games, or text-based simulations, provide an interactive platform for benchmarking LLMs, with the advantages that they match the text modality of LLMs natively, are cheap to implement and extend, and can abstract concepts that would be difficult to depict in visual environments. Building on the classic interactive-fiction line of work, TextWorld (Côté et al., 2018) and Jericho (Hausknecht et al., 2020) established text games as a testbed for sequential decision making, and subsequent benchmarks have used them to probe commonsense reasoning (TextWorld Commonsense (Murugesan et al., 2020)), scientific knowledge (ScienceWorld (Wang et al., 2022)), and general agentic benchmarking and competence (AgentBench (Liu et al., 2024), BALROG (Paglieri et al., 2025)). `Obs2Rule` adopts the same medium for the same reasons, but re-purposes it: rather than presenting the agent with a goal to achieve within the game, each of our environments serves as a laboratory whose only purpose is to be experimented upon, so that the text game becomes an instrument for eliciting rule discovery rather than a task to be solved.

World models as code. Recent research explores whether an agent can express its understanding of an environment as an executable program. WorldCoder (Tang et al., 2024) synthesizes a Python transition function from interaction traces, and Tsividis et al. (2021) recover interpretable rule programs in grid worlds. Crucially, because these world models exist to support planning, an agent need only capture the fragment of the environment its tasks traverse, and the traces it learns from are given in advance or produced incidentally while pursuing a goal.

In short, discovery benchmarks score task completion; symbolic regression evaluates rule induction but supplies the observations; and world-model-as-code work synthesizes verifiable rules in service of task solving. `Obs2Rule` closes the loop, requiring the agent to acquire its own evidence and submit a rule graded on conditions it may never have observed.

3 OBS2RULE

3.1 OBS2RULE ENVIRONMENTS

`Obs2Rule` environments are implemented in Python. Each environment is implemented as a text game that can be viewed as a laboratory in which the agent conducts experiments to uncover the underlying rule governing the environment. Figure 1 shows a playthrough of a typical `Obs2Rule` environment. The agent is initialized in the center of an empty 7×5 grid map.

Actions. Agents interact with the environment through predefined text commands. At each time step the agent is shown the full list of available actions and selects one of them. `Obs2Rule` provides a set of basic actions shared across environments, supplying standard game functionality such as moving around, taking and putting down objects, and opening and closing containers. To let agents design and carry out experiments under a wide range of conditions, `Obs2Rule` additionally implements `create` and `remove`, which allow objects to be introduced into or deleted from the environment at will. In the example of Figure 1, the agent creates a door in order to probe whether teleportation onto a door succeeds when the door is open and when it is closed. Note that this trajectory reflects one experiment the agent designed for itself, and reveals only part of how `teleport` behaves: many conditions remain unprobed, and it is the agent's own responsibility to decide which further experiments are needed.

Objects, Properties, State: All objects in `Obs2Rule` are defined as Python classes inheriting from a base `GameObject` class, which provides each object a `properties` attribute. The *state* in an

`Teleport` Variation 1 (Active Scenario)

Task Description: Figure out the underlying rules of the `teleport` action…

Experiment Design: Teleport to an open door and a closed door.

`Initial Observation:`



`> create door`
The door_uuid_2 is placed in the environment at position (3, 1).

`> look`



`> teleport to 3 1`
You cannot teleport because the target is blocked by other objects.

`> open door_uuid_2`
The door_uuid_2 is now open.

`> look`



`> teleport to 3 1`
The agent has been teleported to (3, 1).

`> look`



`> finish`
You have ended the exploration.

Figure 1: A playthrough of the `teleport` variation 1 environment. The agent receives no in-environment goal and no reward; the task description shown here is an out-of-environment instruction informing the agent that its sole objective is to uncover the rule behind `teleport`. The agent's behavior is guided by an experiment design it produces itself, and this trajectory shows one of the experiments it carries out. Lines beginning with "$>$" are agent actions. Here the agent creates a door, finds that teleporting onto the closed door fails, opens the door, and finds that the same target now succeeds. ***Note:*** *textual grid observations are rendered graphically for readability; the graphical depictions of the environment in this figure were generated by the Seed model, and the raw interface is shown in Appendix A.1.*

environment refers to the collection of all objects and their properties at a certain time step. Game states can be changed by actions, and may also change automatically at each time step. Refer to Section 3.2 for examples of these two kinds of state changes.

Unlike most existing interactive game benchmarks, in which the agent is asked to accomplish a specific objective, `Obs2Rule` environments contain no in-environment tasks and provide no reward for interacting with the environment.

3.2 Scenarios

We design three types of scenarios: active, passive, and stochastic. Together, they mirror the principal modes of inquiry found in empirical science: intervening on a system to learn its response, passively observing how it evolves on its own, and repeated sampling to characterize a stochastic process. In this sense, each scenario type isolates a distinct skill that a scientific agent must master.

Active scenarios. The goal in an active scenario is to determine the underlying rule of a specific action, such `teleport` in the example in Figure 1. The agent must write a function that accurately predicts the state transition resulting from the target action. This is a fundamental skill in scientific experiments, in which the agent must decide which conditions to set up, apply the action, and compare the outcome against its current hypothesis, much as a chemist varies reagents to isolate the mechanism of a reaction.

Passive scenarios. In the real world, most changes occur without any action on our part: a cart slides down on an inclined plane, or an object left on a stove grows hotter. Passive scenarios simulate changes of this kind, requiring the agent to observe and infer how the environment evolves from different states and distill a law from the observed trajectories (e.g. how temperature of objects on the stove changes).

Stochastic scenarios. Stochastic scenarios require the agent to sample observations from the environment in order to estimate a probability. We define two such scenarios: a simple one, in which the agent must estimate the probability of flipping heads with a biased coin, and a complex one, in

which the probability to be predicted depends on several environmental factors. These scenarios test a complementary aspect of scientific practice: real measurements are noisy, so the agent must reason statistically, gather enough samples for its estimate to be reliable, and recognize when an apparent effect is merely sampling variation rather than a genuine dependence on an environmental factor. To reduce the token cost of repeating the same action many times (e.g. flipping a coin), `Obs2Rule` provides a shortcut form of stochastic actions that lets the agent repeat an action n times in a single call, so that its budget is spent on experimental design and hypothesis formation rather than on mechanical repetition.

A full list of scenarios and their descriptions can be found in Appendix A.3.

3.3 Scenario Variations

`Obs2Rule` comprises 12 distinct scenarios, each with 2 to 5 variations, ends up with 32 environments. The variations of a scenario share the same surface setting but differ in the underlying rule that governs it. Consider the `take` scenario, in which the agent must discover the rule behind the `take` action that picks up objects in the environment: one variation may impose an inventory limit of three objects, so that no further object can be taken once the inventory is full, while another imposes no such limit.

Every scenario additionally includes a counter-intuitive variation whose underlying rule violates common sense or established physical law. For instance, a counter-intuitive variation of `take` allows the agent to retrieve an object from a closed container without first opening it. Such variations prevent the agent from succeeding by relying on prior knowledge alone and force it to ground its conclusions in observations gathered from the environment.

3.4 Test cases and Evaluation Metrics

Each `Obs2Rule` environment is paired with human-designed test cases that assess the correctness of the LLM-generated code. Human annotators were asked to enumerate all possible combinations of conditions relevant to the target function, construct a scene realizing each condition, and record the resulting state transition of the environment, or equivalently the inputs and outputs of the target function. This results in 572 test cases across 8 scenarios. Four remaining scenarios have a finite number of possible function inputs so that we can use all possible inputs as test cases.

We report the *pass rate* of each baseline agent on `Obs2Rule`. For active and passive scenarios, an agent passes an environment if its generated code reproduces the output of the ground-truth code exactly on every test case. For stochastic scenarios, where estimation error is unavoidable, the generated code is deemed to pass a test case if its predicted probability lies in the open interval $(r - 0.01, r + 0.01)$, where r denotes the true probability.

4 Experiments and Results

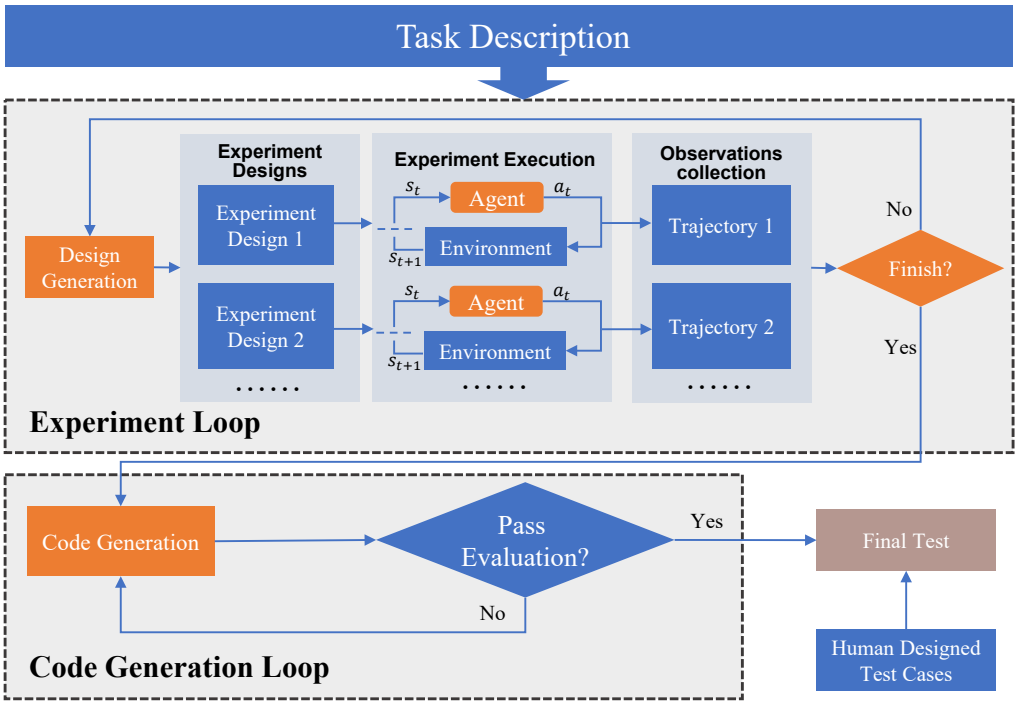
4.1 Baseline Agents

Figure 2(a) shows the design of our baseline agent, which comprises an experiment loop and a code generation loop. We do not build our agents on any existing agent framework. Prompt templates for each of the orange block in Figure 2 can be found in Appendix A.2.

In the experiment loop, the agent designs and conducts experiments in the environment to collect observations. Given the task description, an LLM first proposes a list of experiment designs reflecting its initial guess about how the target functionality works. Each design is some natural language sentences describing what the agent would like to do in the environment. Examples of experiment designs can be found in Figure 2(b). Each design is then executed in the actual environment, and the resulting trajectories are recorded. From these trajectories the agent updates its understanding of the environment and proposes further experiments. The loop terminates once the agent judges that it has collected everything it needs, or once the interaction budget is exhausted.

In the code generation loop, the agent turns the collected trajectories into the target code, a Python function whose role depends on the scenario type. For active scenarios, the function implements

(a)



(b) **Experiment Design Examples:**

```
[
      "Create objects at different distances directly in front of you and teleport to
      see whether distance changes the destination.",
      "Teleport to an open door and a closed door.",
 …]
```

(c) **Target Function Signature Example (`teleport`):**

```
def action(state: dict, x: int, y: int) -> str:
    """
    Simulate the {target_action} action.
    Args:
        state: A dictionary with keys "agent" (Agent instance) and "world" (World
instance).
        x: The x-coordinate of the target position.
        y: The y-coordinate of the target position.
    Returns:
        A string describing the result of the action (observation).
    """
    pass
```

Figure 2: (a) Workflow of the baseline agent; orange blocks denote stages in which the LLM is invoked. (b) Examples of experiment designs generated by the LLM. (c) An example of the target function signature (`teleport`) that the LLM must complete.

a specific action, updating the game state in place and returning an observation string describing the consequence of the action. For passive scenarios, it takes a state and predicts how the relevant object properties change, e.g., how many degrees the temperature of an object on a stove rises. For stochastic scenarios, it returns the target probability, such as the probability of obtaining heads from a coin flip. In every case the agent is supplied with a function signature (see the example in Figure 2(c)) so that the generated function conforms to the evaluation interface.

The generated code is first evaluated on the trajectories the agent collected itself, to verify that it is consistent with the evidence gathered. Whenever inconsistencies arise, the agent revises the code

Table 1: Pass rate of 4 LLMs on `Obs2Rule`

Models	Active Scenarios	Passive Scenarios	Stochastic Scenarios	Overall
DeepSeek	0.53	0.85	0.5	**0.66**
GLM	0.4	0.85	0.5	0.59
Kimi	0.2	0.69	0.5	0.44
Qwen	0.2	0.77	0.5	0.47

using the failing cases together with any error messages. Once the code passes all collected cases or the revision budget is exhausted, the version performing best on the collected data is taken as the final output and evaluated on the full suite of manually designed test cases to obtain its score. For stochastic scenarios the revision stage is skipped, since a predicted probability cannot be checked exactly against a finite set of sampled observations.

We instantiate the agent with four LLMs: Qwen3.7-max, DeepSeek-V4-pro, GLM-5.2, and Kimi K2.6. Commercial APIs for those models are used in experiments. We set temperature to 1 for all LLM calls. The agent may propose an unlimited number of experiments, and each experiment is granted up to 50 steps of interaction in the corresponding environment. Experiments are proposed over at most three rounds before the agent proceeds to the code generation loop, where it is allowed a single round of code revision; preliminary experiments indicated that one revision round suffices for current LLMs to repair their code on our tasks.

4.2 Results

Table 1 reports the pass rate of the four LLM-driven agents on `Obs2Rule`, broken down by scenario type. Our main observations are as follows.

`Obs2Rule` is challenging for LLM-driven agents, and active scenarios are by far the hardest. The best-performing agent, driven by DeepSeek, reaches an overall pass rate of only 0.66. Performance on active scenarios is uniformly low: DeepSeek again leads with 0.53, barely more than half of the environments, while the weakest models pass just 0.20. Passive scenarios are handled far more reliably, with DeepSeek and GLM both reaching 0.85 and even the weakest model, Kimi, reaching 0.69. This may be because in a passive scenario the environment evolves on its own, so merely setting up a state and watching it unfold already yields informative observations; an active scenario, by contrast, requires the agent to decide for itself which interventions to apply and under which conditions. The bottleneck therefore lies in experimental design rather than in the inductive step that follows it.

LLMs can estimate simple probabilities, but fail once a probability depends on multiple conditions. `Obs2Rule` contains two stochastic scenarios, `flip coin` and `capture slime`, each with two variations. All four LLMs pass every `flip coin` environment and fail every `capture slime` environment. `Flip coin` tests probability estimation in its simplest form, and although all models succeed, we observe that they tend to round their estimates to a convenient value rather than report the raw sampled frequency. In `capture slime`, the probability of a successful capture depends on the values of three factors, and no model produced even an approximately correct account of how each factor contributes. The limiting ability is thus not probability estimation itself, but the capacity to control variables and design ablation experiments that isolate the contribution of each factor.

5 Discussion

Do agents succeed by guessing from pretrained commonsense knowledge, or by systematic exploration in active scenarios?

Active scenarios require the agent to generate code capturing the rule of an action in the environment. Actions in `Obs2Rule` modify the environment state in place and return an observation string describing the consequence of the action. The pass rate defined above considers only whether the state is modified correctly, ignoring the correctness of the returned string; we therefore introduce

Table 2: Observation accuracy and case-level pass rate of 4 LLMs on `Obs2Rule` active scenarios, with the environment-level pass rate for comparison

	DeepSeek	**GLM**	**Kimi**	**Qwen**
Pass rate	0.53	0.4	0.2	0.2
Observation Accuracy	0.4	0.2	0.27	0.2
Case level Pass rate	0.91	0.82	0.87	0.88

Table 3: Pass rate of four LLMs on `Obs2Rule` under oracle observations. Numbers in parenthesis are pass rate in Table 1 for comparison. Each agent is given human-collected observations covering every corner case of the environment and asked only to induce the rule, so that performance reflects rule induction alone. The large gain over Table 1, where agents must gather their own evidence, shows that experiment design rather than induction is the bottleneck.

Models	**Active Scenarios**	**Passive Scenarios**	**Stochastic Scenarios**	**Overall**
DeepSeek	0.87(0.53)	0.85(0.85)	0.5(0.5)	0.81(0.66)
GLM	0.67(0.4)	0.92(0.85)	0.5(0.5)	0.75(0.59)
Kimi	0.93(0.2)	0.77(0.69)	0.5(0.5)	0.81(0.44)
Qwen	0.93(0.2)	0.92(0.77)	0.5(0.5)	0.88(0.47)

the *observation accuracy* to capture the latter. A generated function passes the observation test of an environment if it returns exactly the ground truth observation string on every test case, and the observation accuracy is the percentage of environments whose observation test is passed.

This metric is informative because of an asymmetry between the two kinds of output: state changes can often be guessed from commonsense knowledge even when an entire class of conditions is never probed, whereas the exact wording of an observation string is virtually impossible to reproduce without having seen it. The observation accuracy thus acts as a proxy for exploration coverage, largely independent of how well an agent's priors happen to align with the environment.

Table 2 reports observation accuracy of all four models. For the stronger models (DeepSeek and GLM) we find a substantial gap between the two metrics: DeepSeek's generated code passes 53% of environments on state transitions but only 40% on observation strings. These agents therefore inferred correctly what certain state transitions must look like without ever having triggered them, meaning their success rests in part on prior knowledge rather than on evidence they actually gathered. For the weaker models the two rates lie much closer together, but for the opposite reason: these agents fail to get the state transitions right in the first place, so the gap narrows from above rather than because their exploration is more systematic. In neither case does an agent earn its score by thoroughly covering the observation space, which is precisely the behavior `Obs2Rule` is designed to measure.

Does a failed rule reflect a wrong understanding of the environment, or an incomplete one?

The *Case-level pass rate* is the percentage of test cases passed by the generated code within a single environment, which we average over all 15 active scenario environments. Whereas the pass rate is all-or-nothing, this metric provides a finer-grained view of how far a generated rule is from the ground truth.

Table 2 reports case-level pass rate of 4 LLMs. While the environment-level pass rates are only moderate, case-level pass rates are uniformly high. DeepSeek attains the best score of 0.91. This indicates that failures are rarely wholesale: a generated rule is typically correct on the large majority of conditions and breaks on only a small handful of test cases, sometimes just one or two. This pattern reinforces the picture drawn above. **The agents are not fundamentally incapable of expressing environment rules as code; rather, they often miss narrow corners of the condition space.** Because a scientific law must hold universally, however, such near-misses are still failures, and the all-or-nothing pass rate is the metric we regard as primary. The gap between the two numbers quantifies exactly what is lacking: not inductive ability, but the exhaustiveness of the experiments that precede induction.

Table 4: Pass rate of four LLMs on counter-intuitive versus intuitive variations.

	DeepSeek	GLM	Kimi	Qwen
Counter-Intuitive	0.75	0.67	0.5	0.5
Intuitive	0.6	0.55	0.4	0.45

Is the Bottleneck Experiment Design or Rule Induction?

The pass rate evaluates rule discovery as a whole. To disentangle the two stages, we additionally supply each agent with oracle observations (i.e. observations that cover all corner cases) and ask it only to generate the corresponding code, thereby isolating rule induction from experiment design. The manually designed test cases serve naturally as such observations. We verify by hand that agents do not hard code the outputs of the supplied cases, and report the results in Table 3.

Active scenarios improve dramatically, from 0.33 to 0.85 on average, whereas passive scenarios gain only 0.08 and stochastic scenarios not at all. This confirms that what agents lack in active scenarios is evidence rather than inductive ability: given adequate observations, they express the rule correctly. Stochastic scenarios remain at exactly 0.50 for every model in both settings, since no model recovers the three-factor dependence in `capture slime` even from complete data; and no model reaches ceiling under oracle observations, leaving a residual 12-25% of environments that fail for inductive reasons. Experiment design is therefore the dominant bottleneck, but not the only one.

Does agents work worse in counter-intuitive variations?

Every `Obs2Rule` scenario includes one counter-intuitive variation, whose rule contradicts common sense or physical law. Table 4 breaks down the pass rate by counter-intuitive and intuitive variations. Contrary to our expectation, all four agents perform *better* on the counter-intuitive variations, and the effect is consistent across models. Conflict with prior knowledge is therefore not the principal source of difficulty: once an agent observes behavior that contradicts its expectations, it is generally able to faithfully encode what it saw rather than defaulting to its priors. What makes a variation hard is instead its complexity (i.e. the number of factors that contributes to the rule) and the more complex variations in `Obs2Rule` happen to be the intuitive ones, since counter-intuitive rules are typically obtained by negating or removing a single condition from the intuitive default.

One qualification is worth noting: because counter-intuitive variations are by construction simpler than their intuitive counterparts, this comparison confounds intuitiveness with complexity and should not be read as evidence that counter-intuitive rules are inherently easier. Disentangling the two would require variations matched for the number of conditions, which we leave to future work.

6 Conclusion

We introduced `Obs2Rule`, a text-based benchmark of 32 manually designed environments to benchmark rule discovery ability of LLM-driven agents. `Obs2Rule` consists of 12 scenarios in which the agent receives no goal and no reward, must design and conduct its own experiments to uncover a latent rule, and must submit that rule as a Python function graded against human-authored test cases. Scenario variations that look identical but obey different rules, including deliberately counter-intuitive ones, ensure that success cannot be obtained from prior knowledge alone.

Our experiments with four LLM-driven agents show that the best agent solves only approximately two thirds of the benchmark, and that performance is markedly weaker on active scenarios, where the agent must choose its own actions to discover the rule. Our further analysis show that for current agents, the bottleneck of rule discovery is not articulating a law but systematically gathering the evidence needed to identify it. Progress will likely come less from better program synthesis than from agents that reason explicitly about coverage: maintaining competing hypotheses, identifying the conditions that discriminate among them, and recognizing when the evidence so far cannot justify a general claim. We release `Obs2Rule` to make that progress measurable.

AI USE STATEMENT

In this work, we only used generative AI tools for paper polishing and the 4 graphical representations of `Obs2Rule` in Figure 1 are AI generated. All codes, including the `Obs2Rule` environments and experiment scripts, were written by human. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, interpret results. Formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in the writing of proofs are not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f12b4df26344f3be803c06b555252efe-Paper-Conference.pdf.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532, 2018.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Petar Sirkovic, Artiom Myaskovsky, Grzegorz Glowaty, Felix Weissenberger, Alessio Orlandi, Dan Popovici, Anil Palepu, Keran Rong, Ryutaro Tanno, Khaled Saab, Fan Zhang, Jacob Blum, Andrew Carroll, Kavita Kulkarni, Nenad Tomašev, Dina Zverinski, Ivor Rendulic, Elahe Vedadi, Florian Hasler, Luka Rimanic, Marina Boia, Ivan Budiselic, Ben Feinstein, Mathias Bellaiche, Tom Sheffer, Jan Freyberg, Jeremy Ratcliff, Ottavia Bertolli, Katherine Chou, Avinatan Hassidim, Burak Gokturk, Amin Vahdat, Yuan Guan, Vikram Dhillon, Eeshit Dhaval Vaishnav, Byron Lee, Tiago R. D. Costa, José R. Penadés, Gary Peltz, Yossi Matias, James Manyika, Demis Hassabis, Yunhan Xu, Pushmeet Kohli, Annalisa Pawlosky, Alan Karthikesalingam, and Vivek Natarajan. Accelerating scientific discovery with co-scientist. *Nature*, 655(8122):487–496, Jul 2026. ISSN 1476-4687. doi: 10.1038/s41586-026-10644-y. URL https://doi.org/10.1038/s41586-026-10644-y.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. Interactive fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 7903–7910, 2020.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 20271–20309. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/huang24y.html.

Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 10088–10116. Curran Associates, Inc., 2024. doi: 10.52202/079017-0324. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/13836f251823945316ae067350a5c366-Paper-Datasets_and_Benchmarks_Track.pdf.

David Klahr and Kevin Dunbar. Dual space search during scientific reasoning. *Cognitive Science*, 12(1):1–48, 1988. doi: https://doi.org/10.1207/s15516709cog1201_1. URL https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1201_1.

Akshay Krishnamurthy, Keegan Harris, Dylan J. Foster, Cyril Zhang, and Aleksandrs Slivkins. Can large language models explore in-context? In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 120124–120158. Curran Associates, Inc., 2024. doi: 10.52202/079017-3818. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/d951f73c521d069fefbb73396df01424-Paper-Conference.pdf.

William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason Moore. Contemporary symbolic regression methods and their relative performance. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c0c7c76d30bd3dcaefc96f40275bdc0a-Paper-round1.pdf.

Haokun Liu, Yangqiaoyu Zhou, Mingxuan Li, Chenfei Yuan, and Chenhao Tan. Literature meets data: A synergistic approach to hypothesis generation. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 245–281, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.12. URL https://aclanthology.org/2025.acl-long.12/.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/e9df36b21ff4ee211a8b71ee8b7e9f57-Paper-Conference.pdf.

Chris Lu, Cong Lu, Robert Tjarko Lange, Yutaro Yamada, Shengran Hu, Jakob Foerster, David Ha, and Jeff Clune. Towards end-to-end automation of ai research. *Nature*, 651(8107):914–919, Mar 2026. ISSN 1476-4687. doi: 10.1038/s41586-026-10265-5. URL https://doi.org/10.1038/s41586-026-10265-5.

Keerthiram Murugesan, Mattia Atzeni, Pavan Kapanipathi, Pushkar Shukla, Sadhana Kumaravel, Gerald Tesauro, Kartik Talamadupula, Mrinmaya Sachan, and Murray Campbell. Text-based rl agents with commonsense knowledge: New challenges, environments and baselines. In *AAAI Conference on Artificial Intelligence*, 2020. URL https://api.semanticscholar.org/CorpusID:222209017.

Davide Paglieri, Bartł omiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Woł czyk, Akbir Khan, Eduardo Pignatelli, Ł ukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Foerster, Jack Parker-Holder, and Tim Rocktaeschel. Balrog: Benchmarking agentic llm and vlm reasoning on games. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 96666–96702, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f0b1515be276f6ba82b4f2b25e50bef0-Paper-Conference.pdf.

Yusuf H Roohani, Andrew H. Lee, Qian Huang, Jian Vora, Zachary Steinhart, Kexin Huang, Alexander Marson, Percy Liang, and Jure Leskovec. BioDiscoveryAgent: An AI Agent for Designing Genetic Perturbation Experiments. In *International Conference on Learning Representations*, 2025. URL https://mlanthology.org/iclr/2025/roohani2025iclr-biodiscoveryagent/.

Parshin Shojaee, Ngoc-Hieu Nguyen, Kazem Meidani, Amir Barati Farimani, Khoa D Doan, and Chandan K. Reddy. LLM-SRBench: A new benchmark for scientific equation discovery with

large language models. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 55325–55359. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/shojaee25a.html.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 70148–70212. Curran Associates, Inc., 2024. doi: 10.52202/079017-2243. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/820c61a0cd419163ccbd2c33b268816e-Paper-Conference.pdf.

Pedro Tsividis, João Loula, Jake Burga, Sergio Arnaud, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J. Gershman, and Joshua B. Tenenbaum. Human-level reinforcement learning through theory-based modeling, exploration, and planning. *ArXiv*, abs/2107.12544, 2021. URL https://api.semanticscholar.org/CorpusID:236447747.

Silviu-Marian Udrescu and Max Tegmark. Ai feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020. doi: 10.1126/sciadv.aay2631. URL https://www.science.org/doi/abs/10.1126/sciadv.aay2631.

Qingyun Wang, Doug Downey, Heng Ji, and Tom Hope. SciMON: Scientific inspiration machines optimized for novelty. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 279–299, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.18. URL https://aclanthology.org/2024.acl-long.18/.

Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. ScienceWorld: Is your agent smarter than a 5th grader? In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.775. URL https://aclanthology.org/2022.emnlp-main.775/.

A Appendix

A.1 Raw trajectory of the example in Figure 1

Figure 3 presents the raw observation of the example shown in Figure 1.

A.2 Agent prompt templates

Figure 4 shows the template used to construct the experiment generation prompt in the first round, and Figure 5 the template used in subsequent rounds. In both templates, TASK_DESCRIPTION is a placeholder for the environment-specific task description, and PAST_RECORDS for the observations collected in previous rounds of experimentation.

Figure 6 shows the prompt used when the agent is conducting experiments. EXPERIMENT_DESIGN represents one specific experiment design that the current experiment should follow. HISTORY is past action and observations in this experiments. CURRENT_STEP is the current step index, CURRENT_OBSERVATION is the observation at current step. POSSIBLE_ACTIONS represents possible actions that the agent can take at the current step.

Figure 7 shows the routing prompt in the experiment loop. If the agent's response starts with "yes" and less than 3 rounds of experiments have been conducted, the agent will continue to design and conduct more experiments. Otherwise, the agent will move on to generate code.

Figure 8 and Figure 9 show prompts for code generation. OBJ_DEFINITIONS represents the definition of objects in the environment. TARGET represents the action to predict in active scenarios

teleport (active scenario)

Task Description: Figure out the underlying rules of the `teleport` action...

Initial Observation:

```
...
...
(0, 2): empty (1, 2): empty (2, 2): empty (3, 2): the
agent_uuid_1 facing north (4, 2): empty (5, 2): empty (6, 2):
empty
...
...
> create door
The door_uuid_2 is placed at (3, 1).
> look
...
(0, 1): empty (1, 1): empty (2, 1): empty (3, 1):
door_uuid_2(close) (4, 1): empty (5, 1): empty (6, 1): empty
(0, 2): empty (1, 2): empty (2, 2): empty (3, 2): the
agent_uuid_1 facing north (4, 2): empty (5, 2): empty (6, 2):
empty
...
...
> teleport to 3 1
You cannot teleport because the target
is blocked by other objects.
> open door_uuid_2
The door_uuid_2 is now open.
> look
...
(0, 1): empty (1, 1): empty (2, 1): empty (3, 1):
door_uuid_2(open) (4, 1): empty (5, 1): empty (6, 1): empty
(0, 2): empty (1, 2): empty (2, 2): empty (3, 2): the
agent_uuid_1 facing north (4, 2): empty (5, 2): empty (6, 2):
empty
...
...
> teleport to 3 1
The agent has been teleported to (3, 1).
> look
...
(0, 1): empty (1, 1): empty (2, 1): empty (3, 1):
door_uuid_2(open), the agent_uuid_1 facing north (4, 1): empty (5,
1): empty (6, 1): empty
...
...
...
> finish
You have ended the exploration.
```

Figure 3: Raw observations for the example shown in Figure 1. Grid rows containing no objects are elided and marked "...".

and the target function to predict in passive and stochastic scenarios. FUNCTION_SIGNATURE represents the function signature (as is in the example in Figure 2(c)).

Figure 10 shows code reflection prompts if the generated code fail any cases collected by the agent. GENERATED_CODE represents generated code with bugs and ERROR_DESCRIPTION represents descriptions of errors and bugs.

Experiment Design Generation Prompt Template (First Round)

You try to understand some mechanism of a text game and eventually replicate it as a Python function. Specifically, your task is: **TASK_DESCRIPTION**.
You do not have any past experience about the game. Write experiments that you want to conduct in the environment to help you know more about the environment.
Each experiment you design should be written in plain text.
Your response should be in JSON format as follows:

```
“‘JSON
[
    ["experiment 1"],
    ["experiment 2"],
    ...
]
```

Here is an example which lists experiments to test the ‘turn‘ action which will change the direction that the agent face.

```
“‘JSON
[
    ["Try to turn to each direction."],
    ["Turn to the direction that the agent is facing."]
]
“‘
```

Figure 4: Experiment Design Generation Prompt Template (First Round)

Experiment Design Generation Prompt Template (Subsequent Rounds)

You try to understand some mechanism of a text game and eventually replicate it as a Python function. Specifically, your task is: **TASK_DESCRIPTION**.
Here are past observation records you have collected in the environment.
PAST_RECORDS
List experiments you want to conduct so that you can find other corner cases which have not been covered in the records.
Each experiment you design should be written in plain text.
Your response should be in JSON format as follows:

```
“‘JSON
[
    ["experiment 1"],
    ["experiment 2"],
    ...
]
```

Here is an example which lists experiments to test the ‘turn‘ action which will change the direction that the agent face.

```
“‘JSON
[
    ["Try to turn to each direction."],
    ["Turn to the direction that the agent is facing."]
]
“‘
```

Figure 5: Experiment Design Generation Prompt Template (Subsequent Rounds)

A.3 Scenario Descriptions

Table 5 lists all scenarios in `Obs2Rule` along with their type, number of variations, and description.

Experiment Conduction Prompt

You will control a character in a text–based game to complete in–game tasks. The current game task is:
TASK_DESCRIPTION
To accomplish the task, your task is to follow the instruction to conduct an experiment in the environment:
EXPERIMENT_DESIGN
Your output must strictly follow these rules
1. Before executing an action, analyze the current environment and reason about the next step to take.
2. When you decide to execute an action, the action command **must** be placed between a pair of <action> and </action> tags.
For example: 'Based on the current situation, I need to check my inventory <action>inventory</action>'
3. The <action> tags must contain only one valid action from the list of valid actions, and nothing else.
4. When you feel you are done, take the action 'finish' to end the game.
HISTORY
Observation of Step **CURRENT_STEP**: **CURRENT_OBSERVATION**
Available actions:
POSSIBLE_ACTIONS

Figure 6: Experiment Conduction Prompt

Routing Prompt (Experiment Loop)

Your ultimate task is to collect enough information from experiments in a text game and use the collected information to replicate some functionality of the game by writing Python code. Here is a description of the functionality you are currently working on: **TASK_DESCRIPTION**
You have already collected those observation records:
PAST_RECORDS
Based on the observations above, do you want to collect extra observations?
Your response should either be "yes" or "no" with no other words.

Figure 7: Routing Prompt (Experiment Loop)

Table 5: Overview of the 12 `Obs2Rule` scenarios. "# Var." gives the number of variations of each scenario, each governed by a different underlying rule.

Scenario	Type	# Var.	Description
`close`	active	2	Close a door or a container
`move`	active	3	Move the agent through the grid
`open`	active	2	Open a door or a container
`put`	active	2	Put an object down or into a container
`take`	active	3	Pick up an object
`teleport`	active	3	Teleport the agent to a target coordinate
`inclined_plane`	passive	2	Predict the position of a cart sliding down an inclined plane
`phone_battery`	passive	5	Predict the battery consumption of a phone at each time step
`stove`	passive	3	Predict the temperature change of objects placed on a stove
`zombie_movement`	passive	3	Predict how zombies move through the grid
`capture_slime`	stochastic	2	Predict the probability of capturing a slime with different equipment
`flip_coin`	stochastic	2	Predict the probability of heads when flipping a biased coin

Code Generation Prompt (Active Scenarios)

Task
You are tasked with writing a Python function that simulates some functionality in a text–based game.
The detailed task description is: **TASK_DESCRIPTION**
Here are the definition of objects in the game environment:
‘‘‘python
OBJ_DEFINITIONS
‘‘‘
Input State and Function Signature
The function receives a ‘state‘ dictionary and action–specific arguments. The ‘state‘ dictionary has the following structure:
‘‘‘python
state = {{
”agent”: <Agent instance>,
”world”: <World instance>,
}}
‘‘‘
The agent is an instance of the Agent class (which inherits from Container and GameObject). Key attributes and methods:

agent.properties: A dictionary containing:
”pos_x” (int): The agent’s x–coordinate on the grid.
”pos_y” (int): The agent’s y–coordinate on the grid.
”facing” (str): The direction the agent is facing (”north”, ”south”, ”east”, ”west”).
”isMoveable” (bool): Whether the agent can be moved.
”isPassable” (bool): Whether the agent can be passed through.
”isContainer” (bool): Whether the agent can contain objects (always True).
agent.contains: A list of GameObject instances the agent is carrying.
agent.get_property(name): Returns the value of a property, or None if not found.
agent.set_property(name, value): Sets a property value (only if the key already exists).
agent.add_object(obj): Adds an object to the agent’s inventory.
agent.remove_object(obj): Removes an object from the agent’s inventory.
World Instance
The world is an instance of the World class (which inherits from GameObject). Key attributes and methods:
world.width (int): The width of the game grid.
world.height (int): The height of the game grid.
world.grid: A 2D list [x][y] where each cell contains a list of GameObject instances at that position.
world.get_objects_at_location(x, y): Returns a list of objects at position (x, y).
world.add_object(obj, pos_x, pos_y): Places an object at the given position.
world.remove_object(obj): Removes an object from the world.
world.agent: Reference to the agent in the world.

The function signature for the ‘**TARGET**‘ action is as follows:
‘‘‘python
FUNCTION_SIGNATURE
‘‘‘
Example Function
Here is an example of another action function for reference. Your implementation should follow a similar structure and style:
‘‘‘python
EXAMPLE_FUNCTION
‘‘‘
Observed State Transitions
The following state transitions were collected by executing the **TARGET** action in the game environment under different conditions. Use these observations to infer the rules of the action. Note that while the observations provided are in the JSON format, your implementation should work with the actual Agent and World instances, not their JSON representations.
PAST_RECORDS

Requirements
– Implement the action function that reproduces the behavior observed above.
– The function must return an observation string describing what happened. Your function should generate observation strings exactly the same as the observations in the state transitions shown above.
– The function must modify the state in–place (e.g., updating agent position, properties, etc.) consistent with the observed transitions.
– Do NOT import any external libraries. Only use standard Python and the methods available on the state objects.
– Provide ONLY the function code. Do not include explanations, tests, or any other text outside the function.
– Put your code inside ‘‘‘python ‘‘‘ chunk so that it can be parsed.

Figure 8: Code Generation Prompt (Active Scenarios)

Code Generation Prompt (Passive & Stochastic Scenarios)

Task
You are tasked with writing a Python function that simulates some functionality in a text–based game.
The detailed task description is: **TASK_DESCRIPTION**
Here are the definition of objects in the game environment:
“‘python
OBJ_DEFINITIONS
“‘
The function signature for the ‘**TARGET**‘ action is as follows:
“‘python
FUNCTION_SIGNATURE
“‘
Observed State Transitions
The following state transitions were collected by executing the **TARGET** action in the game environment under different conditions. Use these observations to infer the rules of the action. Note that while the observations provided are in the JSON format, your implementation should work with the actual Agent and World instances, not their JSON representations.
PAST_RECORDS

Requirements
– Implement the action function that reproduces the behavior observed above.
– Do NOT import any external libraries. Only use standard Python and the methods available on the state objects.
– Provide ONLY the function code. Do not include explanations, tests, or any other text outside the function.
– Put your code inside “‘python “‘ chunk so that it can be parsed.

Figure 9: Code Generation Prompt (Passive & Stochastic Scenarios)

Reflection Prompt

You are tasked with writing a Python function that simulates part of a text–based game. Here is your task:
TASK_DESCRIPTION
Here is the code you have written for this task:
GENERATED_CODE
However, your code could not pass the test cases below:
ERROR_DESCRIPTION
Requirements
– Fix your code so that it reproduces the behavior observed above.
– The function must return an observation string describing what happened. Your function should generate observation strings exactly the same as the observations in the state transitions shown above.
– The function must modify the state in–place (e.g., updating agent position, properties, etc.) consistent with the observed transitions.
– Do NOT import any external libraries. Only use standard Python and the methods available on the state objects.
– Provide ONLY the function code. Do not include explanations, tests, or any other text outside the function.
– Put your code inside “‘python “‘ chunk so that it can be parsed.

Figure 10: Reflection Prompt