Agentic Property-Based Testing: Finding Bugs Across the Python Ecosystem

Muhammad Maaz
MATS, Anthropic

Liam DeVoe
Northeastern University

Zac Hatfield-Dodds
Anthropic

Nicholas Carlini
Anthropic

Abstract

Property-based testing (PBT) is a lightweight formal method, typically implemented as a randomized testing framework. Users specify the input domain for their test using combinators supplied by the PBT framework, and the expected properties or invariants as a unit-test function. The framework then searches for a counterexample, e.g. by generating inputs and calling the test function. In this work, we demonstrate an LLM-based agent which analyzes Python modules, infers function-specific and cross-function properties from code and documentation, synthesizes and executes PBTs, reflects on outputs of these tests to confirm true bugs, and finally outputs actionable bug reports for the developer. We perform an extensive evaluation of our agent across 100 popular Python packages. Of the bug reports generated by the agent, we found after manual review that 56% were valid bugs and 32% were valid bugs that we would report to maintainers. We then developed a ranking rubric to surface high-priority valid bugs to developers, and found that of the 21 top-scoring bugs, 86% were valid and 81% we would report. The bugs span diverse failure modes from serialization failures to numerical precision errors to flawed cache implementations. We reported 5 bugs, 4 with patches, including to NumPy and cloud computing SDKs, with 3 patches merged successfully. Our results suggest that LLMs with PBT provides a rigorous and scalable method for autonomously testing software. Our code and artifacts are available at: `https://github.com/mmaaz-git/agentic-pbt`.

1 Introduction

Property-based testing (PBT) is a software testing paradigm that aims to verify whether a general property holds over a predefined input domain. In contrast to traditional example-based testing, it does not require specific examples of expected outputs. Instead, PBT allows the developer to define invariants that should hold for all inputs, which are then checked on a diverse set of automatically-generated inputs. A property might require that the output of a function is always non-negative, that some f and g commute for all x ($f(g(x)) = g(f(x))$), or that an interpreted program has equivalent semantics to its compiled form. Popularized by QuickCheck in Haskell (Claessen and Hughes, 2000), there now exist PBT libraries for many of the common programming languages, including Python's *Hypothesis* (MacIver et al., 2019), the focus in this work. PBT can be more robust than example-based testing which inherently relies on the developer to anticipate edge cases (MacIver, 2019).

Despite its theoretical appeal, property-based testing is less popular than example-based testing due to the challenge of identifying meaningful properties to test (Goldstein et al., 2024). Specifying meaningful properties, especially as codebases get more complex, often requires significant domain expertise and time investment. Recent advances in large language models (LLMs) have demonstrated remarkable code understanding, which makes it possible to *automatically* mine for properties.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: The 4th Deep Learning for Code Workshop.

However, existing work on using LLMs for PBTs focuses on a single function at a time, and has only a single generation step.

In our work, we introduce *agentic property-based testing*, an approach leveraging the multi-step reasoning capabilities of coding agents and the rigor of PBT. We develop an agent that can autonomously crawl through and understand entire codebases, look for high-value properties (including across functions and classes), write PBTs and run them, and then analyze failing tests for validity. After the agent runs for several rounds, it either surfaces a bug, or states it is unable to identify one.

We demonstrate the effectiveness of agentic PBT through a large-scale evaluation across the Python ecosystem, testing 100 popular Python packages with billions of downloads. Manual evaluation of the bug reports generated by our agent showed that it successfully identified genuine bugs in several widely-used libraries: for example, our agent autonomously finds a bug in NumPy, the pre-eminent numerical library in Python with more than 500 million monthly downloads, which was acknowledged by the maintainers, showing the real-world applicability of our approach. This work represents the largest systematic evaluation of AI-driven property-based testing to date, establishing a new paradigm for scalable software auditing.

2 Agent

Our agent is built on top of Anthropic's Claude Code, a terminal-based coding agent (Anthropic, 2025b), that allows Claude to execute bash commands and read/write files. Our agent is implemented as a natural language prompt, stored in a Markdown file, that is passed to Claude Code. As it is a prompt, our agent is easily portable to other agent implementations, like OpenAI's Codex OpenAI (2025), or general frameworks like ReAct (Yao et al., 2022). Our main contribution is our comprehensive evaluation which demonstrates that LLM-based PBT works at scale, finding diverse bugs with few false alarms.

Our agent is built to work with Python codebases, and generates Hypothesis PBTs to test that code. It takes a single argument, which points the agent towards a particular target, either a single Python file (e.g., `normalizers.py`), module (e.g., `numpy`, `scipy.signal`), or function (e.g., `requests.get()`, `json.loads()`).

We developed the prompt by using industry best-practices (Anthropic, 2025a), collating high-value information from the Hypothesis documentation (Hypothesis Contributors, 2025), and iterative refinement from manual inspection of agent runs. The full prompt is in Appendix A.

Prompt Overview The agent is given the following six instructions, paraphrased below:

1. Analyze the target: Figure out if the target is a module, function, or a file.
2. Understand the target: Find and read documentation, function signatures, source code, etc.
3. Propose properties: Look for properties grounded in the information from the past step. Some examples of high-value properties, e.g., invariants, round-trip, metamorphic, are given. Thoroughly understand the input domain, e.g., by looking at functions that call the function under test.
4. Write tests: Translate these properties into Hypothesis property-based tests.
5. Execute and triage tests: Run tests with `pytest`. For failing tests, reflect using a rubric. If the failing test is a false alarm, go to Step 4 and refine the testing strategy. For passing tests, verify that the test is meaningful.
6. Report bugs: If convinced a bug is genuine, create a bug report in a Markdown file, following a standard format including a summary of the bug, the PBT that exposed the bug, a short reproducing script, why it is a bug, and possibly a proposed patch.

Finally, we provide the agent with a short reference to Hypothesis, which shows essential patterns, some key testing principles, and links to online documentation which can be fetched if needed.

Key Design Choices Our agent maintains a to-do list to track its progress through the 6-step cycle, as well as to keep track of, e.g., which properties it would like to test. We attempt to reduce false alarms throughout by emphasizing that proposed properties should be strongly supported by evidence, asking the agent to reflect on failing tests carefully, and to focus on a few high-value properties. Lastly, we let the agent have broad autonomy: it decides which functions to target, and decides when a failing property is truly a bug (with some direction from a rubric).

3 Experiment

We evaluate our agents ability to find bugs by curating a diverse corpus of 100 Python packages. We combine three complementary sampling approaches to ensure comprehensive coverage:

- *Hand-selected from the standard library ($n = 15$):* `json`, `pathlib`, `collections`, `itertools`, `functools`, `datetime`, `re`, `os`, `urllib`, `statistics`, `decimal`, `base64`, `uuid`, `html`, `csv`.
- *Hand-selected from third-party PyPI packages ($n = 15$):* `numpy`, `pandas`, `requests`, `flask`, `sqlalchemy`, `matplotlib`, `scipy`, `beautifulsoup4`, `pydantic`, `fastapi`, `django`, `tornado`, `keras`, `httpie`, `black`.
- *Random sample from the top 5,000 PyPI packages, by downloads ($n = 70$):* Variety of domains including data, cloud computing, and machine learning. The full list is provided with our code.

For each package, we test the agent on the main module (e.g., for `beautifulsoup4`, the main module is `bs4`), and all submodules one level deep (e.g., `numpy.linalg`). In total, we ran on 933 modules.

We used Claude Opus 4.1 Anthropic (2025c) as the LLM for all agent runs. Each agent was run in an isolated virtual environment containing the package under test and its dependencies, along with Hypothesis. Agents were run in parallel, with up to $N = 20$ concurrently. Agents had read/write permission within their working directory, could execute `python` and `pytest` bash commands, had read access to the virtual environment for source code, and internet access. The experiment was run on a RunPod Ubuntu 20.04 container with 8 vCPUs, 16 GB RAM, and 50 GB disk.

Evaluation Definitive validation of reported bugs across such diverse libraries is difficult as it requires domain expertise in each specific codebase. First, we developed an initial scoring rubric with the goal of eliminating clear false alarms and got Claude Opus 4.1 to grade all reports. We randomly sampled $n = 50$ bug reports from the top 80% by score. Two of the authors independently scored each bug report, using two criteria: "Is this a valid bug? If yes, would we reasonably report this to the maintainers?". Disagreements were discussed to arrive at a final consensus. Using insights from the initial rubric, we developed a final rubric (see our code) which scores bug reports out of 15, got Claude Opus 4.1 to grade all reports, and manually reviewed all reports with a $15/15$ score. To validate our agents with maintainers with actual domain expertise, we (manually) reported some particularly interesting bugs to their respective repositories.

4 Results

Summary statistics We evaluated our agent across 100 Python packages covering 933 modules. The agent generated 984 bug reports, discovering issues in 786 modules (84.2%) and averaging just over one bug per module. Total agent runtime was 136.6 hours (82 min/package, 8.8 min/module) and total API cost was $5,474.20 ($54.74/package, $5.87/module, $5.56/bug report). Individual module costs ranged from $0.65 to $15.42, with a median of $5.81. Each agent run involved roughly 110 turns. In aggregate, the agents used 2.21 billion (input & output) tokens (2.37 million/module).

Manual review Inter-rater agreement on the initial review was $\kappa = 0.31$. After reaching consensus, 56.0% (95% CI: 42.2%, 69.8%) of reports were determined to be valid bugs, and 32.0% (95% CI: 19.1%, 44.9%) were valid and worth reporting. With the final rubric, 18 of the 21 top-scoring reports were valid, and 17 of those were both valid and worth reporting.

Reported bugs While examining reports, we selected the following particularly interesting bugs to report to the original developer. The corresponding bug reports generated by the agent are in Appendix B. They demonstrate the range of issues discoverable through our approach:

[numpy] (numpy.random)
Property: Wald distribution should only return non-negative samples.
Bug: Negative samples sometimes returned due to catastrophic cancellation.
Status: Bug acknowledged. Patch merged. [PR #29609]

[aws-lambda-powertools] (aws_lambda_powertools.shared.functions)
Property: `slice_dictionary()` splits a dictionary into chunks which should be able to be recombined into the original dictionary.

Bug: It returns the same (first) chunk repeatedly, due to not incrementing the iterator.
Status: Bug acknowledged. Patch merged. [PR #7246]

[cloudformation-cli-java-plugin] (rpdk.core.jsonutils.utils)
Property: `item_hash()` function should produce different outputs for different inputs.
Bug: All list inputs hash to $hash(\texttt{None})$, due to use of the in-place `.sort()` method, which returns `None`.
Status: Patch submitted. [PR #1106]

[tokenizers] (tokenizers.tools)
Property: `EncodingVisualizer.calculate_label_colors()` should return a valid HSL color format.
Bug: The returned string is missing a closing parenthesis.
Status: Bug acknowledged. Patch merged. [PR #1853]

[python-dateutil] (dateutil)
Property: `easter()` should be on a Sunday.
Bug: Returns a non-Sunday date for the Julian calendar in various years.
Status: Issue reported. Maintainers identified the behavior as intended due to differing calendar systems, and acknowledged the semantics as confusing. [Issue #1437]

5 Discussion and Conclusion

Our evaluation demonstrates that LLM-guided property-based testing can systematically uncover bugs missed by traditional testing. With a cost of $5.56/bug report, and extrapolating from our manual grading that 56% of these are valid bugs, our agent can find bugs for $9.93/valid bug. This is an upper bound on the real-world cost, where developers with domain expertise can be more judicious with where to target the agent. The diversity of issues, spanning numerical issues to business logic issues, show the power of PBT, and the ability of agents to autonomously mine for such properties.

Limitations The primary limitation is that we did not manually review all 984 bug reports. However, our review of a subset of reports shows that the false discovery rate has a 95% CI of [30.2%, 57.8%]. The secondary limitation is intent ambiguity: the agent cannot distinguish intentional design violations from bugs. An example of this is a bug report about `LookupDict` from the `requests` library: based on its name and the fact it subclasses from the built-in `dict`, our agent tested whether it exhibits `dict`-like behavior, which it in fact does not. The same issue was raised on GitHub in 2022[1], but the maintainer replied that it was not meant to work like `dict`. In practice, both limitations can be mitigated by developers' domain expertise rather than blind exhaustive testing as we did.

Comparison to Related Work Vikram et al. (2023) study converting library documentation into Hypothesis PBTs for 40 functions across 10 popular Python libraries (e.g., `numpy`). With their best model and prompting strategy, only 41% of generated PBTs ran without error and passed on the implemented code, and at most 21% of documented properties were captured. In contrast, we do better by using an agentic approach, which is better able to capture more complex properties due to multiple steps. More recent work He et al. (2025) propose a different take on this problem: a "generator" LLM produces candidate code while a "tester" LLM synthesizes PBTs from the problem description. PBT failures are fed back to guide code refinement, yielding improvements over example-based test-driven development. There is also extensive work on LLMs for software testing in general: see Wang et al. (2024) for a review.

Conclusion We presented an automated approach for bug discovery using LLM-guided property-based testing, which discovered real bugs across several popular Python libraries. The properties and failing test cases discovered show the utility of combining LLMs with PBT. While such a tool is useful for developers, a possible malicious use is autonomous discovery of vulnerabilities. As LLMs improve and get cheaper, agentic PBT could become an increasingly valuable complement to traditional testing, helping developers systematically explore code behavior and find bugs.

[1] `https://github.com/psf/requests/issues/6238`

References

Anthropic. Claude Code: Best practices for agentic coding. https://www.anthropic.com/engineering/claude-code-best-practices, Apr. 2025a. Accessed: 2025-08-20.

Anthropic. Claude Code. https://www.anthropic.com/claude-code, 2025b. Accessed: 2025-08-20.

Anthropic. System Card Addendum: Claude Opus 4.1. Technical report, Anthropic, Aug 2025c. URL https://assets.anthropic.com/m/4c024b86c698d3d4/original/Claude-4-1-System-Card.pdf.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

L. He, Z. Chen, Z. Zhang, J. Shao, X. Gao, and L. Sheng. Use property-based testing to bridge llm code generation and validation. *arXiv preprint arXiv:2506.18315*, 2025.

Hypothesis Contributors. Hypothesis documentation. https://hypothesis.readthedocs.io/en/latest/, 2025. Accessed: 2025-08-20.

D. MacIver. In praise of property-based testing. *URL: https://increment. com/testing/in-praise-of-property-based-testing*, 2019.

D. R. MacIver, Z. Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

OpenAI. OpenAI Codex. https://openai.com/codex/, 2025. Accessed: 2025-08-20.

V. Vikram, C. Lemieux, J. Sunshine, and R. Padhye. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346*, 2023.

J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4): 911–936, 2024.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629. Accessed: 2025-08-20.

A Agent prompt

Note: some characters may not render properly in the PDF; we provide the full prompt file in our code release.

```
---
description: Property-based testing agent
---

# Property-Based Testing Bug Hunter

You are a **bug-hunting agent** focused on finding genuine bugs through property-based testing with Hypothesis. Your mission: discover real bugs by testing fundamental properties that should always hold.

## Your Todo List

Create and follow this todo list for every target you analyze:

1. [ ] **Analyze target**: Understand what you're testing (module, file, or function)
2. [ ] **Understand the target**: Use introspection and file reading to understand implementation
3. [ ] **Propose properties**: Find evidence-based properties the code claims to have
4. [ ] **Write tests**: Create focused Hypothesis tests for the most promising properties
5. [ ] **Test execution and bug triage**: Run tests with `pytest` and apply bug triage rubric to any failures
6. [ ] **Report or conclude**: Either create a bug report or report successful testing

Mark each item complete as you finish it. This ensures you don't skip critical steps.
You can use the `Todo` tool to create and manage your todo list.
Use the `Todo` tool to keep track of properties you propose as you test them.

## Core Process

Follow this systematic approach:

### 1. Analyze target
- Determine what you're analyzing from `$ARGUMENTS`:
  - Empty → Explore entire codebase
  - `.py` files → Analyze those specific files
  - Module names (e.g. `numpy` or `requests`) → Import and explore those modules
  - Function names (e.g. `numpy.linalg.solve`) → Focus on those functions
  ```bash
  python -c "import numpy; print('success - treating as module')"
  python -c "from numpy import abs; print(type(numpy.abs))"
  ```

2. Understand the target

Use Python introspection to understand the module or function you are testing.

To find the file of a module, use `target_module.__file__`.

To get all public functions/classes in the module, use `inspect.getmembers(target_module)`.

To get the source code of a function, signature, and docstring, of a function `func` use:
- `inspect.signature(func)` to get the signature
- `func.__doc__` to get the docstring
- `inspect.getsource(func)` to get the source code

To get the file of a function, use `inspect.getfile(target_module.target_function)`.

You can then use the Read tool to read full files.

If explicitly told to test a file, you **must use** the Read tool to read the full file.

Once you have the file location, you can explore the surrounding directory structure with `os.path.dirname(target_module.__file__)` to understand the module better.
You can use the List tool to list files, and Read them if needed.

Sometimes, the high-level module just imports from a private implementation module.
Follow those import chains to find the real implementation, e.g., `numpy.linalg._linalg`.

Together, these steps help you understand:
- The module's structure and organization
- Function information, including signature and docstring
- Entire code files, so you can understand the target in context, and how it is called
- Related functionality you might need to test
- Import relationships between files

3. Propose properties

Once you thoroughly understand the target, look for these high-value property patterns:

- **Invariants**: `len(filter(x)) <= len(x)`, `set(sort(x)) == set(x)`
- **Round-trip properties**: `decode(encode(x)) = x`, `parse(format(x)) = x`
- **Inverse operations**: `add/remove`, `push/pop`, `create/destroy`
- **Multiple implementations**: fast vs reference, optimized vs simple
- **Mathematical properties**: idempotence `f(f(x)) = f(x)`, commutativity `f(x,y) = f(y,x)`
- **Confluence**: if the order of function application doesn't matter (eg in compiler optimization passes)
- **Metamorphic properties**: some relationship between `f(x)` and `g(x)` holds, even without knowing the correct value for `f(x)`. For example, `sin(π - x) = sin(x)` for all x.
```

````
- **Single entry point**: for libraries with 1-2 entrypoints, test that calling it on valid inputs doesn't crash (no specific property!). Common in e.g. parsers.

If there are no candidate properties in $ARGUMENTS, do not search outside of the specified function, module, or file. Instead, exit with "No testable properties found in $ARGUMENTS".

**Only test properties that the code is explicitly claiming to have.** either in the docstring, comments, or how other code uses it. Do not make up properties that you merely think are true. Proposed properties should be **strongly supported ** by evidence.

**Function prioritization**: When analyzing a module/file with many functions, focus on:
- Public API functions (those without leading underscores) with substantive docstrings
- Multi-function properties, as those are often more powerful
- Single-function properties that are well-grounded
- Core functionality rather than internal helpers or utilities

**Investigate the input domain** by looking at the code the property is testing. For example, if testing a function or class, check its callers. Track any implicit assumptions the codebase makes about code under test, especially if it is an internal helper, where such assumptions are less likely to be documented. This investigation will help you understand the correct strategy to write when testing. You can use any of the commands and tools from Step 2 to help you further understand the codebase.

### 4. Write tests

Write focused Hypothesis property-based tests to test the properties you proposed.

- Use smart Hypothesis strategies - constrain inputs to the domain intelligently
- Write strategies that are both:
  - sound: tests only inputs expected by the code
  - complete: tests all inputs expected by the code
  If soundness and completeness are in conflict, prefer writing sound but incomplete properties. Do not chase completeness: 90% is good enough.
- Focus on a few high-impact properties, rather than comprehensive codebase coverage.

A basic Hypothesis test looks like this:

```python
@given(st.floats(allow_nan=False, min_value=0))
def test_sqrt_round_trip(x):
    result = math.sqrt(x)
    assert math.isclose(result * result, x)
```

A more complete reference is available in the *Hypothesis Quick Reference* section below.

### 5. Test execution and bug triage

Run your tests with `pytest`.

**For test failures**, apply this bug triage rubric:

**Step 1: Reproducibility check**
- Can you create a minimal standalone reproduction script?
- Does the failure happen consistently with the same input?

**Step 2: Legitimacy check**
- Does the failing input represent realistic usage?
  - ✓ Standard user inputs that should work
  - ✗ Extreme edge cases that violate implicit preconditions
- Do callers of this code make assumptions that prevent this input?
  - Example: If all callers validate input first, testing unvalidated input is a false alarm
- Is the property you're testing actually claimed by the code?
  - ✓ Docstring says "returns sorted list" but result isn't sorted
  - ✗ Mathematical property you assumed but code never claimed

**Step 3: Impact assessment**
- Would this affect real users of the library?
- Does it violate documented behavior or reasonable expectations?

**If false alarm detected**: Return to Step 4 and refine your test strategy using `st.integers(min_value=...)`, `strategy.filter(...)`, or `hypothesis.assume(...)`. If unclear, return to Step 2 for more investigation.

**If legitimate bug found**: Proceed to bug reporting.

**For test passes**, verify the test is meaningful:
- Does the test actually exercise the claimed property?
  - ✓ Test calls the function with diverse inputs and checks the property holds
  - ✗ Test only uses trivial inputs or doesn't actually verify the property
- Are you testing the right thing?
  - ✓ Testing the actual implementation that users call
  - ✗ Testing a wrapper or trivial function that doesn't contain the real logic

### 6. Bug Reporting

Only report **genuine, reproducible bugs**:
- ✓ "Found bug: `json.loads(json.dumps({"??": None}))` fails with KeyError"
- ✓ "Invariant violated: `len(merge(a,b)) != len(a) + len(b)` for overlapping inputs"
- ✗ "This function looks suspicious" (too vague)
- ✗ False positives from flawed test logic

**If genuine bug found**, categorize it as one of the following:
- **Logic**: Incorrect results, violated mathematical properties, silent failures
````

````
- **Crash**: Valid inputs cause unhandled exceptions
- **Contract**: API differs from its documentation, type hints, etc

And categorize the severity of the bug as one of the following:
- **High**: Incorrect core logic, security issues, silent data corruption
- **Medium**: Obvious crashes, uncommon logic bugs, substantial API contract violations
- **Low**: Documentation, UX, or display issues, incorrect exception type, rare edge cases

Then create a standardized bug report using this format:

````markdown
# Bug Report: [Target Name] [Brief Description]

**Target**: `target module or function`
**Severity**: [High, Medium, Low]
**Bug Type**: [Logic, Crash, Contract]
**Date**: YYYY-MM-DD

## Summary

[1-2 sentence description of the bug]

## Property-Based Test

```python
[The exact property-based test that failed and led you to discover this bug]
```

**Failing input**: `[the minimal failing input that Hypothesis reported]`

## Reproducing the Bug

[Drop-in script that a developer can run to reproduce the issue. Include minimal and concise code that reproduces the issue, without extraneous details. If possible, reuse the mininal failing input reported by Hypothesis. **Do not include comments or print statements unless they are critical to understanding**.]

```python
[Standalone reproduction script]
```

## Why This Is A Bug

[Brief explanation of why this violates expected behavior]

## Fix

[If the bug is easy to fix, provide a patch in the style of `git diff` which fixes the bug, without commentary. If it is not, give a high-level overview of how the bug could be fixed instead.]

```diff
[patch]
```

````

File naming: Save as `bug_report_[sanitized_target_name]_[timestamp]_[hash].md` where:
- Target name has dots/slashes replaced with underscores
- Timestamp format: `YYYY-MM-DD_HH-MM` using `datetime.now().strftime("%Y-%m-%d_%H-%M")`
- Hash: 4-character random string using `''.join(random.choices(string.ascii_lowercase + string.digits, k=4))`
- Example: `bug_report_numpy_abs_2025-01-02_14-30_a7f2.md`

7. **Outcome Decision**
- **Bug(s) found**: Create bug report file(s) as specified above - you may discover multiple bugs!
- **No bugs found**: Simply report "Tested X properties on [target] - all passed ✓" (no file created)
- **Inconclusive**: Rare - report what was tested and why inconclusive

Hypothesis Quick Reference

Essential Patterns
```python
import math

from hypothesis import assume, given, strategies as st


# Basic test structure
@given(st.integers())
def test_property(x):
    assert isinstance(x, int)


# Safe numeric strategies (avoid NaN/inf issues)
st.floats(allow_nan=False, allow_infinity=False, min_value=-1e10, max_value=1e10)
st.floats(min_value=1e-10, max_value=1e6)  # positive floats

# Collection strategies
st.lists(st.integers())
st.text()


# Filtering inputs
@given(st.integers(), st.integers())
```
```

````
def test_division(a, b):
    assume(b != 0)  # Skip when b is zero
    assert abs(a % b) < abs(b)
```

### Key Testing Principles
- Use `math.isclose()` or `pytest.approx()` for float comparisons
- Focus on properties that reveal genuine bugs when violated
- Use `@settings(max_examples=1000)` to increase testing power
- Constrain inputs intelligently rather than defensive programming
- Do not constrain strategies unnecessarily. Prefer e.g. `st.lists(st.integers())` to `st.lists(st.integers(), max_size=100)`, unless the code itself requires `len(lst) <= 100`.

### Documentation Resources

For a comprehensive reference:

- **Basic tutorial**: https://hypothesis.readthedocs.io/en/latest/quickstart.html
- **Strategies reference**: https://hypothesis.readthedocs.io/en/latest/reference/strategies.html
- **NumPy strategies**: https://hypothesis.readthedocs.io/en/latest/reference/strategies.html#numpy
- **Pandas strategies**: https://hypothesis.readthedocs.io/en/latest/reference/strategies.html#pandas

### Rare but useful strategies

These strategies are uncommon, but highly useful where relevant.

- `st.from_regex`
- `st.from_lark` - for context-free grammars
- `st.functions` - generates arbitrary callable functions

Use the WebFetch tool to pull specific documentation when needed.

---

If you generate files in the course of testing, leave them instead of deleting them afterwards. They will be automatically cleaned up after you.

**Remember**: Your goal is finding genuine bugs, not generating comprehensive test suites. Quality over quantity. One real bug discovery > 100 passing tests.

Now analyze the targets: $ARGUMENTS
````
````

B Selected bug reports

A selection of bug reports, as written by the agent.

B.1 numpy

````
# Bug Report: numpy.random.wald Produces Negative Values

**Target**: `numpy.random.wald`
**Severity**: High
**Bug Type**: Logic
**Date**: 2025-08-18

## Summary

The `numpy.random.wald` function produces negative values when called with large mean parameters (>= 1e8), violating the mathematical definition of the Wald (inverse Gaussian) distribution which only produces positive values.

## Property-Based Test

```python
import numpy.random
from hypothesis import given, strategies as st, settings

@given(
    st.floats(min_value=1e8, max_value=1e15),
    st.floats(min_value=0.1, max_value=10.0)
)
@settings(max_examples=50)
def test_wald_negative_values(mean, scale):
    """Wald distribution should never produce negative values"""
    samples = numpy.random.wald(mean, scale, size=1000)
    assert all(s >= 0 for s in samples), f"Found negative values with mean={mean}, scale={scale}"
```

**Failing input**: `mean=100000000.0, scale=1.099609375`

## Reproducing the Bug

```python
import numpy.random

numpy.random.seed(42)
mean = 1e8
scale = 1.0
samples = numpy.random.wald(mean, scale, size=1000)

negative_samples = samples[samples < 0]
print(f"Generated {len(samples)} samples")
print(f"Found {len(negative_samples)} negative values")
print(f"Minimum value: {samples.min()}")
```

## Why This Is A Bug

The Wald (inverse Gaussian) distribution is mathematically defined to only produce positive values. The probability density function is only defined for x > 0, and the documentation states that the function "Draw[s] samples from a Wald, or inverse Gaussian, distribution" with "mean : float or array_like of floats - Distribution mean, must be > 0".

Negative values violate this fundamental property and can cause downstream issues in applications that rely on the mathematical properties of the distribution, such as reliability modeling, financial modeling, and statistical inference.

## Fix

The issue likely stems from numerical precision problems when computing the inverse transform or rejection sampling for large mean values. The implementation should include bounds checking to ensure all generated values are strictly positive, possibly by:

1. Adding explicit bounds checking after sample generation
2. Using a more numerically stable algorithm for large mean values
3. Switching to an alternative sampling method when mean exceeds a certain threshold

```diff
# Conceptual fix in the sampling routine
def wald_sample(mean, scale):
    # ... existing sampling logic ...
    sample = compute_sample(mean, scale)
+   # Ensure non-negativity due to numerical issues
+   if sample < 0:
+       sample = 0.0  # or resample
    return sample
```
````

B.2 aws-lambda-powertools

````
# Bug Report: aws_lambda_powertools.shared.functions.slice_dictionary Returns Duplicate Chunks

**Target**: `aws_lambda_powertools.shared.functions.slice_dictionary`
**Severity**: High
**Bug Type**: Logic
**Date**: 2025-08-18

## Summary

The `slice_dictionary` function incorrectly returns the same chunk multiple times instead of properly slicing the dictionary into different parts, causing data loss and duplication.

## Property-Based Test

```python
from hypothesis import given, strategies as st
from aws_lambda_powertools.shared.functions import slice_dictionary

@given(
    data=st.dictionaries(
        keys=st.text(min_size=1, max_size=20),
        values=st.one_of(st.integers(), st.text(), st.booleans()),
        min_size=0,
        max_size=100
    ),
    chunk_size=st.integers(min_value=1, max_value=20)
)
def test_slice_dictionary_confluence(data, chunk_size):
    """Test that sliced dictionary chunks can be reconstructed to the original."""
    chunks = list(slice_dictionary(data, chunk_size))

    # Reconstruct the dictionary from chunks
    reconstructed = {}
    for chunk in chunks:
        reconstructed.update(chunk)

    assert reconstructed == data, f"Reconstruction failed: {data} != {reconstructed}"
```

**Failing input**: `data={'0': 0, '00': 0}, chunk_size=1`

## Reproducing the Bug

```python
import sys
sys.path.insert(0, '/root/hypothesis-llm/envs/aws-lambda-powertools_env/lib/python3.13/site-packages')
from aws_lambda_powertools.shared.functions import slice_dictionary

# Example 1: Lost keys
data = {'0': 0, '00': 0}
chunks = list(slice_dictionary(data, chunk_size=1))
print(f"Original: {data}")
print(f"Chunks: {chunks}")
# Output: Chunks: [{'0': 0}, {'0': 0}]
# Expected: [{'0': 0}, {'00': 0}]

# Example 2: All chunks are identical
data = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
chunks = list(slice_dictionary(data, chunk_size=2))
print(f"Original: {data}")
print(f"Chunks: {chunks}")
# Output: [{'a': 1, 'b': 2}, {'a': 1, 'b': 2}, {'a': 1, 'b': 2}]
# Expected: [{'a': 1, 'b': 2}, {'c': 3, 'd': 4}, {'e': 5}]
```

## Why This Is A Bug

The function is supposed to split a dictionary into chunks of the specified size, but instead it repeatedly yields the same first `chunk_size` items. This violates the expected behavior of:
1. Each key appearing exactly once across all chunks
2. Chunks being different slices of the original dictionary
3. Being able to reconstruct the original dictionary from the chunks

## Fix

```diff
--- a/aws_lambda_powertools/shared/functions.py
+++ b/aws_lambda_powertools/shared/functions.py
@@ -141,5 +141,8 @@ def powertools_debug_is_set() -> bool:


 def slice_dictionary(data: dict, chunk_size: int) -> Generator[dict, None, None]:
-    for _ in range(0, len(data), chunk_size):
-        yield {dict_key: data[dict_key] for dict_key in itertools.islice(data, chunk_size)}
+    it = iter(data)
+    for _ in range(0, len(data), chunk_size):
+        chunk_keys = list(itertools.islice(it, chunk_size))
+        if chunk_keys:
+            yield {dict_key: data[dict_key] for dict_key in chunk_keys}
```
````

B.3 cloudformation-cli-java-plugin

````
# Bug Report: rpdk.core.jsonutils.utils.item_hash All Lists Hash to Same Value

**Target**: `rpdk.core.jsonutils.utils.item_hash`
**Severity**: High
**Bug Type**: Logic
**Date**: 2025-08-18

## Summary

The `item_hash` function in `rpdk.core.jsonutils.utils` contains a critical bug that causes all list inputs to hash to the same value (the MD5 hash of "null"), completely breaking the hash function's purpose for list-type data.

## Property-Based Test

```python
from hypothesis import given, strategies as st
from rpdk.core.jsonutils.utils import item_hash

@given(st.lists(st.integers()), st.lists(st.integers()))
def test_item_hash_different_lists_different_hashes(list1, list2):
    """Different lists should produce different hashes (except in rare collisions)."""
    if list1 != list2:
        hash1 = item_hash(list1)
        hash2 = item_hash(list2)
        # This test fails because all lists hash to the same value
        assert hash1 != hash2 or list1 == list2
```

**Failing input**: Any two different lists, e.g., `[1, 2, 3]` and `[4, 5, 6]`

## Reproducing the Bug

```python
import sys
sys.path.insert(0, '/root/hypothesis-llm/envs/cloudformation-cli-java-plugin_env/lib/python3.13/site-packages')

from rpdk.core.jsonutils.utils import item_hash

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = ["a", "b", "c"]
empty_list = []

hash1 = item_hash(list1)
hash2 = item_hash(list2)
hash3 = item_hash(list3)
hash4 = item_hash(empty_list)

print(f"item_hash([1, 2, 3]) = {hash1}")
print(f"item_hash([4, 5, 6]) = {hash2}")
print(f"item_hash(['a', 'b', 'c']) = {hash3}")
print(f"item_hash([]) = {hash4}")

assert hash1 == hash2 == hash3 == hash4 == "37a6259cc0c1dae299a7866489dff0bd"
print("BUG: All lists hash to the same value!")
```

## Why This Is A Bug

The `item_hash` function is supposed to generate unique hashes for different inputs. However, due to a coding error on line 32, all list inputs produce the same hash value. This completely defeats the purpose of a hash function, which should produce different outputs for different inputs (except for rare collisions). The bug causes:

1. **Loss of uniqueness**: All lists, regardless of content, produce identical hash values
2. **Hash collisions**: Any code using this for deduplication or caching will fail
3. **Security implications**: If used for any security-sensitive hashing, this would be a critical vulnerability

## Fix

```diff
--- a/rpdk/core/jsonutils/utils.py
+++ b/rpdk/core/jsonutils/utils.py
@@ -29,7 +29,8 @@ def item_hash(
     if isinstance(item, dict):
         item = {k: item_hash(v) for k, v in item.items()}
     if isinstance(item, list):
-        item = [item_hash(i) for i in item].sort()
+        hashed_items = [item_hash(i) for i in item]
+        item = sorted(hashed_items)
     encoded = json.dumps(item, sort_keys=True).encode()
     dhash.update(encoded)
     return dhash.hexdigest()
```

The bug occurs because `.sort()` returns `None`, not the sorted list. The fix uses `sorted()` which returns the sorted list, or assigns the sorted list after calling `.sort()` on it.
````

B.4 tokenizers

````
# Bug Report: tokenizers.tools.visualizer Missing Closing Parenthesis in HSL Color Format

**Target**: `tokenizers.tools.visualizer.EncodingVisualizer.calculate_label_colors`
**Severity**: Medium
**Bug Type**: Contract
**Date**: 2025-08-18

## Summary

The `calculate_label_colors` method generates malformed HSL color strings missing a closing parenthesis, causing CSS parsing errors when used in HTML visualization.

## Property-Based Test

```python
from hypothesis import given, strategies as st
import re
from tokenizers.tools import Annotation, EncodingVisualizer

@given(st.lists(st.text(min_size=1, max_size=20), min_size=1, max_size=10))
def test_calculate_label_colors_valid_hsl_format(labels):
    annotations = [Annotation(i*10, i*10+5, label) for i, label in enumerate(labels)]
    colors = EncodingVisualizer.calculate_label_colors(annotations)

    hsl_pattern = re.compile(r'^hsl\(\d+,\d+%,\d+%\)$')  # Correct HSL format with closing paren
    for label, color in colors.items():
        assert hsl_pattern.match(color), f"Invalid HSL format: '{color}'"
```

**Failing input**: Any non-empty list of annotations, e.g., `['label1']`

## Reproducing the Bug

```python
import sys
sys.path.insert(0, '/root/hypothesis-llm/envs/tokenizers_env/lib/python3.13/site-packages')
from tokenizers.tools import Annotation, EncodingVisualizer

annotations = [Annotation(0, 5, "test_label")]
colors = EncodingVisualizer.calculate_label_colors(annotations)
color = colors["test_label"]

print(f"Generated color: '{color}'")
assert color.endswith(')'), f"Missing closing parenthesis in HSL: {color}"
```

## Why This Is A Bug

The HSL color format in CSS requires the format `hsl(hue, saturation%, lightness%)` with balanced parentheses. The current implementation produces `hsl(10,32%,64%` without the closing parenthesis, which will cause CSS parsing errors when these colors are used in the HTML visualization. This violates the expected HSL format contract and will break the visualization feature.

## Fix

```diff
--- a/tokenizers/tools/visualizer.py
+++ b/tokenizers/tools/visualizer.py
@@ -175,7 +175,7 @@ class EncodingVisualizer:
         colors = {}

         for label in sorted(labels):  # sort so we always get the same colors for a given set of labels
-            colors[label] = f"hsl({h},{s}%,{l}%"
+            colors[label] = f"hsl({h},{s}%,{l}%)"
             h += h_step
         return colors
```
````

B.5 python-dateutil

````
# Bug Report: dateutil.easter Violates Date Range Invariant

**Target**: `dateutil.easter`
**Severity**: High
**Bug Type**: Logic
**Date**: 2025-08-18

## Summary

The `easter.easter()` function returns dates outside the valid Easter range (March 22 - April 25) for certain years and methods, violating fundamental calendar constraints.

## Property-Based Test

```python
from hypothesis import given, strategies as st
````
````

````
from dateutil import easter

@given(st.integers(min_value=1583, max_value=4099))
def test_easter_date_invariants(year):
    """Test that Easter always falls in March or April and on Sunday"""
    for method in [1, 2, 3]:
        try:
            easter_date = easter.easter(year, method)
            # Easter must be in March or April
            assert easter_date.month in [3, 4]
            # Easter must be on Sunday (weekday() == 6)
            assert easter_date.weekday() == 6
        except Exception:
            pass
```

**Failing input**: `year=2480`

## Reproducing the Bug

```python
from dateutil import easter

year = 2480

orthodox_easter = easter.easter(year, method=2)
print(f"Orthodox Easter {year}: {orthodox_easter}")
print(f"Month: {orthodox_easter.month} (should be 3 or 4)")
print(f"Weekday: {orthodox_easter.weekday()} (should be 6 for Sunday)")

julian_easter = easter.easter(year, method=1)
print(f"\nJulian Easter {year}: {julian_easter}")
print(f"Weekday: {julian_easter.weekday()} (should be 6 for Sunday)")
```

Output:
```
Orthodox Easter 2480: 2480-05-05
Month: 5 (should be 3 or 4)
Weekday: 6 (should be 6 for Sunday)

Julian Easter 2480: 2480-04-19
Weekday: 4 (should be 6 for Sunday)
```

## Why This Is A Bug

1. **Method 2 (Orthodox)** returns May 5, 2480, which is outside the valid Easter date range. Easter, by definition, can only fall between March 22 and April 25 in the Gregorian calendar.

2. **Method 1 (Julian)** returns April 19, 2480, which falls on a Friday (weekday=4) instead of Sunday (weekday=6). Easter, by definition, always occurs on Sunday.

These violations break fundamental calendar constraints that users rely upon. The documentation states these methods are valid for years 1583-4099, but the calculations produce invalid results within this range.

## Fix

The bug appears to be in the algorithm implementation for certain edge cases. The fix would require:

1. Validating the calculated date falls within the valid Easter range
2. Ensuring the result is always a Sunday
3. Potentially adjusting the algorithm for problematic years

A defensive fix could include validation:

```diff
--- a/dateutil/easter.py
+++ b/dateutil/easter.py
@@ -70,6 +70,16 @@ def easter(year, method=EASTER_WESTERN):
         g = year % 19
         e = (11*g + 20 + z - x) % 30
         # ... rest of calculation ...
+
+    # Validate the result
+    result = date(int(year), int(month), int(day))
+
+    # Easter must be on Sunday
+    if result.weekday() != 6:
+        raise ValueError(f"Calculated Easter date {result} is not on Sunday")
+
+    # Easter must be in March or April
+    if result.month not in [3, 4]:
+        raise ValueError(f"Calculated Easter date {result} is not in March or April")

     return date(int(year), int(month), int(day))
```
````
````

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: Abstract and results reflect the results.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations in the discussion about evaluation and developer intent.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: No theory.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Agent and experimental setup are stated in the paper. Supplementary materials contains full details and reproduction scripts.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: Prompts and full code attached.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Key experimental setting details are in the paper; full details in the code.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: 95% CI for proportions are stated.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Server details and parallel worker setup described in the paper.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: It does.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Mentioned societal impact.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to

generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: No such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: Does not use existing assets.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: Code and data released with documentation.

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: No human subjects involved.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: No human subjects involved.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are used as part of the methodology of the paper, which is described.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.