Empirical Software Engineering (2027) 32:30
https://doi.org/10.1007/s10664-026-10953-w

Property-based testing in Python: empirical insights

Isadora de Oliveira[1] · **Arthur Lisboa Corgozinho**[1] · **Henrique Rocha**[2] · **Marco Tulio Valente**[1]

Received: 18 September 2025 / Accepted: 10 August 2026

Abstract
Property-Based Testing (PBT) automatically generates test inputs to validate properties of programs, shifting developers' effort from writing examples to specifying invariants. While the technique has gained popularity in Python through the Hypothesis framework, little is known about how developers adopt and use it in practice. This paper reports on three empirical studies. First, we analyzed 367 PBTs from 244 Python projects, classifying them into nine property categories and quantifying their use of Hypothesis constructs. We found that *Test Oracle* properties dominate (29.97%), and that PBTs are generally concise (median 14 LOC), relying heavily on built-in strategies (75.20%), but also on external (22.62%) and internal (17.17%) ones. Second, we studied 213 Stack Overflow posts tagged with PBT, revealing that the main challenges developers face concern data generation strategies (36.62%), especially for composite and tabular data (24.36%). Finally, we evaluated Ghostwriter, Hypothesis's automated test generator, against 203 tests from our dataset; only 18.23% were fully automatable, while most required partial adaptation (30.05%) or were incompatible (51.72%). Together, our findings provide the largest empirical characterization of PBT in Python to date, highlight developers' difficulties in adopting the technique, and expose limitations of current tool support.

Keywords Property-based testing · Python · Hypothesis · Ghostwriter · Stack overflow · Developer adoption or practices · Automated test generation

Communicated by: Dietmar Pfahl.

✉ Marco Tulio Valente
mtvalente@gmail.com

Isadora de Oliveira
isadoraoliveira@dcc.ufmg.br

Arthur Lisboa Corgozinho
arthurlisboa@dcc.ufmg.br

Henrique Rocha
hsrocha@loyola.edu

[1] Department of Computer Science, UFMG, Belo Horizonte, Brazil

[2] Department of Computer Science, Loyola University Maryland, Baltimore, MD 21210, USA

Published online: 24 August 2026

1 Introduction

Automated testing has become a cornerstone of modern software development to ensure better efficiency and reliability when checking software quality, as software testing is commonly used throughout the development process to identify defects and establish confidence in program correctness (Valente 2024; Kracht et al. 2014). Automated tests are predominantly guided by examples (Winters et al. 2020). For instance, when implementing a unit test, a developer first selects some values (i.e., examples) to call the function under test. Then, she calls the function using these values and checks whether the returned results match the expected ones. Even modern testing techniques such as Mutation Testing (Vercacmmen et al. 2023), Test Amplification (Abdi et al. 2022), and Exploratory Test (Coutinho et al. 2023) rely on example-based tests. Despite being widely used, example-based tests require effort and expertise from developers for selecting the most effective test inputs, which is not a simple task. Thus, the idea behind Property-Based Testing (PBT) is to free developers from manually selecting test inputs (Chen et al. 2022; Claessen and Hughes 2000; Fink and Bishop 1997; MacIver et al. 2019). Instead, these values are generated by the testing framework. The ultimate goal is to allow developers to spend more time fixing bugs than defining test case inputs. A common motto among PBT practitioners summarizes this goal: "test faster, fix more."

Therefore, when using PBT, we do not verify a specific result of the function under test, since we do not know the exact input values used by the test, which are automatically generated by the testing framework. Instead, developers implement tests that check properties that should hold for any possible input value. A simple example is shown in Listing 1, assuming a function `add(x,y)`. The first property-based test verifies the commutative property of addition, that is, the result of `add(x,y)` should be the same as the one returned by `add(y,x)`. The second test verifies the neutral element property, that is, `add(x,0)` should always return `x`, for any input `x`. Since the tests are called hundreds or even thousands of times by the testing framework, using randomly generated inputs, the assumption is that existing bugs may be detected in some of such calls.

Listing 1 A simple example of Property-Based Testing (PBT)

```
1  from hypothesis import given, strategies as st
2
3  def add(x,y):
4      return x + y
5
6  @given(x=st.integers(), y=st.integers())
7  def test_add_commutative(x, y):
8      assert add(x, y) == add(y, x)
9
10 @given(x=st.integers())
11 def test_add_zero(x):
12     assert add(x, 0) == x
```

Property-based testing was first proposed for functional languages more than 20 years ago (Claessen and Hughes 2000), along with tool support—most notably QuickCheck—to facilitate the implementation of these tests in programming languages such as Haskell. More recently, PBT has gained traction in Python, mainly due to the popularity of the Hypothesis

framework (MacIver et al. 2019). As of January 2026, the Hypothesis repository on GitHub has more than 8,000 stars. The tool also has at least 200K daily downloads on the PyPI package manager.[1] In a survey conducted by JetBrains with more than 30,000 Python developers in 2024, it ranked sixth among the most widely used Python testing frameworks, being mentioned by 4% of respondents.[2] The frameworks with the highest number of mentions are aimed at unit testing, such as pytest (53%) and unittest (23%). Therefore, at least according to this data, Hypothesis currently represents the leading framework for implementing property-based testing in Python.

Existing Studies: Despite this growing interest, there is still limited empirical evidence about how developers adopt and use property-based testing in practice. A notable exception is the study by Goldstein et al. (2024), which investigated PBT adoption through 30 interviews with developers at Jane Street. Their work provides valuable qualitative insights into the benefits, limitations, and opportunities of property-based testing in an industrial setting. However, because it focuses on a single organization, several important questions remain unanswered. In particular, we still lack a broader understanding of which properties developers most frequently verify, the challenges they face when adopting PBT, and whether existing tool support adequately matches real-world testing practices. Moreover, the study focuses on OCaml, a functional programming language with a specific user community. Consequently, little is known about property-based testing practices in a broader and more representative software ecosystem.

Objective: We intend to investigate PBT adoption in Python, a widely used programming language, by characterizing three complementary dimensions of the technique: development practices, developer challenges, and tool support. To this end, we combine evidence from open-source repositories, developer discussions, and automated tool evaluation to provide a comprehensive empirical understanding of PBT adoption in practice.

Methods: To achieve the proposed objective, we organized our research into three complementary empirical studies. The first analyzes 367 real-world property-based tests to characterize current development practices. The second analyzes 213 Stack Overflow questions to investigate the challenges developers face when adopting PBT. The third evaluates Ghostwriter using 203 real-world property-based tests to assess the effectiveness of current automated test generation. Together, these studies provide complementary evidence on three fundamental dimensions of PBT adoption: development practices, developer challenges, and tool support.

The first study substantially extends our previous short paper (Corgozinho et al. 2023) by expanding the dataset from 86 to 367 property-based tests (an increase of more than 300%) and improving the reliability of the analysis through independent classification by two evaluators. This larger dataset enabled us to identify two previously unreported PBT categories—Testing for Exceptions and Testing for Crashes—and to provide the largest empirical characterization to date of the Hypothesis constructs used in real-world Python projects. These results establish a quantitative baseline for PBT usage in Python and provide practical guidance on the properties and framework constructs most relevant to developers adopting the technique.

[1] https://pypistats.org/packages/hypothesis

[2] https://lp.jetbrains.com/python-developers-survey-2024/

While the first study characterizes development practices, the second investigates the challenges developers face after adopting PBT. Specifically, we analyze 213 Stack Overflow questions related to the Hypothesis framework. We found that more than one third of the questions (36.62%) concern input data generation strategies, with composite data structures (such as tuples, dictionaries, and similar structures) representing the most frequent source of difficulty (37.18%). These findings identify concrete opportunities for improving framework documentation, tutorials, examples, and data-generation APIs, while also helping newcomers anticipate the challenges they are most likely to encounter when implementing their first property-based tests.

Finally, understanding current practices and developer challenges naturally raises a third question: are existing tools capable of supporting the property-based tests developers actually write? To answer this question, we evaluate Ghostwriter, the automatic test generation tool distributed with the Hypothesis framework, using a dataset of 203 real-world tests belonging to property categories that are, in principle, supported by the tool. Our results reveal a substantial gap between current automation capabilities and real-world practice: only 37 tests (18%) could be generated automatically, while another 61 tests (30%) required manual adaptation. These findings highlight important limitations of current template-based generation and motivate future research on more powerful automated and AI-assisted techniques for generating property-based tests.

Contribution: By combining evidence from real-world code, developer discussions, and automated tool evaluation, we provide a comprehensive empirical view of property-based testing adoption in Python. Our study shows that Test Oracle is the most common category of property-based tests (29.97%), that input data generation strategies are the primary challenge faced by developers (36.62% of Stack Overflow questions), and that current automated support remains limited, with Ghostwriter fully generating only 18% of the analyzed tests. Together, these findings provide actionable guidance for developers adopting PBT, empirical evidence to guide the evolution of frameworks and automated tools, and a quantitative baseline for future research.

Structure: The remainder of this article is organized as follows. Section 2 provides an overview of Property-Based Testing, describing the categories of properties that can be verified in this type of testing. We also introduce the Ghostwriter tool, highlighting its role in the automatic generation of tests. Section 3 presents the Characterization Study, which aims to answer two research questions: (RQ1) What are the most common categories of PBT? and (RQ2) What are the most commonly used built-in constructs in Hypothesis? Section 4 describes the Stack Overflow Study, in which we analyze questions asked by developers about PBT (RQ3). Our goal is to identify the main challenges faced in practice when using Hypothesis. This analysis helps us understand recurring difficulties and provides insights into the gaps between PBT theory and its real-world use in software development. In Section 5, we evaluate the Ghostwriter tool, investigating its ability to automatically generate property-based tests (RQ4). Section 6 then discusses the Lessons Learned from our studies, synthesizing the main insights obtained during the analysis and evaluation. Finally, Section 7 presents related work, and Section 8 concludes the paper with final considerations and suggestions for future work.

2 Background

2.1 Property-Based Tests

Property-Based Testing (PBT) is a randomized automated testing technique that can be used to check whether key properties of a function are valid within a given input domain (Claessen and Hughes 2000; Löscher and Sagonas 2017). Unlike example-based testing, PBT relies on a set of automatically generated input values to assert the properties of a function under test. Thus, the focus shifts from checking function results produced using manually selected inputs to verifying function properties under random inputs (MacIver et al. 2019).

A PBT testing tool executes the tests multiple times with random inputs based on an input generator strategy. It also checks whether a given function property—specified by the test developer—holds for such inputs. If this property does not hold, the tool attempts to shrink the failing inputs to simpler cases to make the failure easier to understand and reproduce. Hypothesis (MacIver et al. 2019) is an open-source tool for implementing property-based tests in Python. We selected Hypothesis for our study because it is a robust tool that includes all the key constructs needed to implement PBTs in Python, which is nowadays one of the dominant programming languages in domains such as machine learning and artificial intelligence.

Example 1 In Hypothesis, the `@given` decorator is used to specify the inputs that will be automatically generated for a test case (line 3). This decorator requires a data generation parameter, referred to as a *strategy*. Below, we show an example that uses a strategy to generate random integer parameters (line 3), which are then used to assert the associative property of addition (lines 5–7):

```
1 from hypothesis import given, strategies as st
2
3 @given(a=st.integers(), b=st.integers(), c=st.
      integers())
4 def test_associativity(a, b, c):
5     result1 = (a + b) + c
6     result2 = a + (b + c)
7     assert result1 == result2
```

Example 2 In this example, we deliberately introduce a bug in the `add` function. The failure occurs whenever the `x` parameter is greater than 10,000 (lines 4–5).

```
1  from hypothesis import given, strategies as st
2
3  def add(x,y):
4      if (x > 10000): # bug
5          return x - y
6      return x + y
7
8  @given(x=st.integers(), y=st.integers())
9  def test_add_commutative(x, y):
10     assert add(x, y) == add(y, x)
```

Thus, Hypothesis detects the following failure in this test:

```
1 Falsifying example: test_add_commutative(
2 E x=10001,
3 E y=1,
4 E )
```

The failure occurs because `add(10001,1)` subtracts the input values (due to our bug) and therefore returns 10,000.

2.2 Categories of Properties

Table 1 summarizes the key types of properties that can be checked by PBTs. Five categories are described in an online document cited by the Hypothesis documentation,[3] another—*Outputs Within Expected Bounds*—is mentioned in an article by Hatfield-Dodds (2020). Additionally, *Idempotence* is discussed in the context of Ghostwriter and the remaining two—*Testing for Exceptions* and *Testing for Crashes*—were proposed by us, based on our experience during this study.

Next, we present a more detailed description of each category:

There and Back Again (aka Roundtrip): This category applies when a function produces a result that is immediately accessed and returned by another function (e.g., writing a value to a file and then reading it back). Thus, a distinctive feature of this category is the combination of an operation with its inverse, which should return the same value as the initial input. Another example is encoding a string and then decoding it.

Test Oracle: This category applies when a well-known algorithm is used as an oracle to check the implementation of a novel or untested algorithm (e.g., testing a new sorting algorithm against a default implementation such as QuickSort).

Table 1 Categories of properties used in Property-Based Testing

Category	Description
There and Back Again	Applying an operation and its inverse returns the original value.
Test Oracle	Compare results against a trusted implementation.
Outputs Within Expected Bounds	Verify that outputs remain within expected bounds.
Different Paths, Same Destination	Different execution paths produce the same result.
Some Things Never Change	Verify that invariants are preserved after a transformation.
Idempotence	Repeated application produces the same result as a single application.
Hard to Prove, Easy to Verify	Verify results that are easier to check than compute.
Testing for Exceptions	Verify that invalid inputs raise the expected exception.
Testing for Crashes	Ensure the program does not crash on random inputs.

[3] https://fsharpforfunandprofit.com/posts/property-based-testing-2/

Outputs Within Expected Bounds: This category involves verifying whether the results of a function stays within expected limits. These bounds can be either computational or logical (e.g., if a function returns a price, the value should be non-negative).

Different Paths, Same Destination: This category applies when a combination of functions should produce the same result regardless of the order in which they are applied (e.g., a commutative property).

Some Things Never Change: This category applies when an invariant is preserved between input and output of a transformation (e.g., the size of a list should not change after sorting).

Idempotence: This category applies when repeated applications of an operation preserve the result, producing the same outcome as a single application (e.g., f(f(x)) = f(x)), ensuring stability.

Hard to Prove, Easy to Verify: This category applies when a result can be easily verified, even if the method for computing it is more complex (e.g., checking the output of a tokenization algorithm is simpler than implementing the algorithm itself, since verification only requires concatenating the returned tokens).

Testing for Exceptions: This category includes tests that check whether the function under test throws a specific exception when called with invalid inputs or abnormal conditions.

Testing for Crashes: This category includes tests executed with random inputs for the sole purpose of checking whether the system crashes.

2.3 Ghostwriter Tool

Ghostwriter is a tool designed to assist in writing tests with Hypothesis.[4] Its sole purpose is to generate a template of test functions, thus simplifying the adoption of Property-Based Tests in software projects. The tool can generate tests for the following categories: *Roundtrip*; *Test Oracle*; *Different Paths, Same Destination*; *Idempotence*; and *Testing for Exceptions*.

For example, to generate tests for the Roundtrip category, testers should use the `roundtrip()` function. Suppose a program includes `encoded` and `decoded` functions for converting a string into bytes and then decoding it back. By calling:

```
hypothesis.extra.Ghostwriter.roundtrip(encoded, decoded)
```

Ghostwriter generates the following test:

```
1  @given(text=st.text())
2  def test_roundtrip_encoded_decoded(text):
3      bytes = encoded(text)
4      value = decoded(bytes)
5      assert text == value
```

To keep the paper self-contained, we describe the code templates generated by Ghostwriter for the remaining categories in Appendix A.

[4] https://hypothesis.readthedocs.io/en/latest/reference/integrations.html#ghostwriter

3 Characterization Study

In this section, we present an initial study aimed at characterizing the use of PBT in real-world Python projects. We seek to answer two research questions:

RQ1: *What are the most common categories of PBT?* This question is important because it can guide the adoption of PBT by revealing the properties most frequently tested. For example, developers can begin implementing PBTs for functions in their systems that exhibit these same properties. In other words, since such properties are more common, identifying suitable functions tends to be easier.

RQ2: *What are the most commonly used built-in constructs in Hypothesis?* As with RQ1, the goal is to support developers in taking their first steps toward adopting PBT. By built-in constructs, we mean those provided by Hypothesis, such as strategies (`@given`) and other decorators. We aim to identify the constructs most commonly used in practice. This allows developers to focus on a core subset of constructs, rather than learning the entire set supported by Hypothesis.

3.1 Dataset

To answer the proposed RQs, we retrieved a list of projects that depend on Hypothesis using the `github-dependents-info` feature from GitHub.[5] This feature provides information about GitHub repositories that depend on a given project. In the case of Hypothesis, the resulting list included 367 repositories. However, after an initial inspection, we found projects that import Hypothesis but do not use the library in any file. To discard such false positives, we implemented a Python script to search for real usages of Hypothesis in the selected repositories. For this task, we used the PyGithub[6] and GitPython[7] libraries to retrieve the repositories' URLs and clone them, respectively.

As a result, we selected 244 repositories that contain code depending on Hypothesis. Our final dataset includes widely used projects such as PyTorch (a machine learning framework), NumPy (a scientific computing package), and Polars (a dataframe library). In these 244 projects, we identified 7,612 property-based tests. However, since our analysis required an extensive manual effort, we chose to evaluate a random sample of 367 test cases (covering 108 of the 244 projects).The sample size was computed using the standard statistical procedure for estimating population proportions, assuming a 95% confidence level and a 5% margin of error.

3.2 Study Design

To answer RQ1, we followed the process described in Fig. 1. Essentially, two authors independently and manually classified the 367 tests selected for the study —more than four times the number considered in our preliminary study. This classification was conducted in two rounds. In the first round, the authors focused on classifying the tests using only three popular categories from our initial study: *Roundtrip*, *Test Oracle*, and *Outputs Within*

[5] https://github.com/nvuillam/github-dependents-info

[6] https://pygithub.readthedocs.io

[7] https://gitpython.readthedocs.io

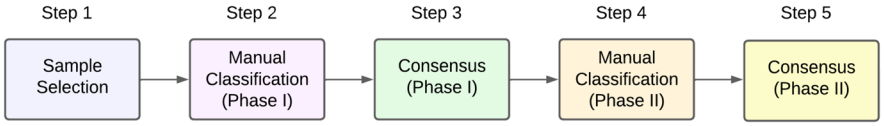

Fig. 1 Methodology used to answer RQ1

Expected Bounds. Tests that did not fit into any of these categories were labeled as *Other*. We chose to begin with popular and more easily identifiable categories to foster a shared understanding of PBTs and the proposed classification criteria between the two authors. The authors then met to compare their results, discuss disagreements, and reach a consensus. In total, 177 tests were classified into the initially selected categories. In the second round, the authors followed a similar process to classify the remaining tests into the other categories, again working independently and holding a second and final consensus meeting. The Kappa coefficient[8] computed after the two rounds of classification was 0.54, indicating a moderate level of agreement between the authors. However, this result can be considered satisfactory given the complexity involved in analyzing real-world tests developed by third parties, which often depend on multiple functions and modules. It is also worth noting that the consensus steps required at least 20 hours of discussion across both rounds.

After completing the analysis of RQ1, we also conducted an exploration of the size of the PBTs in our dataset, measured in lines of code (LOC). This analysis was performed by category and is reported in Section B of the appendix. The idea was to provide a first quantitative characterization of our results.

To address RQ2, the first author conducted a careful analysis of all tests in our dataset. The goal was to identify the most frequently used built-in constructs from Hypothesis. Specifically, the author searched for and counted the usage of the following constructs:

- *strategies* (such as `@integers`, `@floats`, and `@text()`), which are used to define how test inputs are generated. For example, `@given(x=integers(), y=integers())` generates a wide range of integer pairs to test a function like `add(x, y)`.
- *@settings*, which is used to configure test parameters, such as the number of times a test should be executed. For example, `@settings(max_examples=200)` increases the number of generated test cases from the default value to 200.
- *assume()*, which is used to specify conditions that input values must satisfy. For example, `assume(y != 0)` can be used to skip test cases where `y` is zero in a division operation.
- *@example*, which is used to ensure that a specific input example is always tested. For example, `@example(x=0, y=1)` guarantees that the test is executed with `x = 0` and `y = 1`, regardless of the generated data.
- *note()*, which is used to add extra information when a failure occurs. For example, `note(f"Result of divide(x, y): result")` can help track the output that led to the failure.*Summary of methodological changes compared to our previous study* (Corgozinho et al. 2023): we expanded the dataset from 86 to 367 property-based

[8] Cohen's Kappa coefficient measures the level of agreement between two raters while accounting for agreement that could occur by chance (Fleiss and Cohen 1973). It ranges from -1 to 1, where 0 indicates chance-level agreement and higher values indicate stronger agreement.

tests (an increase of over 300%); whereas in the short paper the classification of properties was conducted by a single classifier, in this work the properties are evaluated by two independent classifiers, thereby improving the reliability of the classification. Additionally, the analysis of Stack Overflow posts related to PBTs (Section 4) and of the effectiveness of Ghostwriter for automatically generating PBTs (Section 5) are entirely new.

3.3 What are the Most Common Properties Checked in PBT? (RQ1)

Table 2 shows the classification results for the 367 PBTs. The most common category is *Test Oracle*, accounting for nearly 30% of the tests. It is followed by *Some Things Never Change* (17.44%), *Hard to Prove, Easy to Verify* (16.08%), and *Roundtrip* (13.90%). The four least common categories are *Testing for Exceptions* (4.90%), *Testing for Crashes* (4.36%), *Outputs Within Expected Bounds* (4.36%) and *Idempotence* (a category for which we could not find any PBT instance).

Next, we present examples of PBTs from each of the three most popular categories in our study. First, in the Listing 2, we show an example of *Test Oracle*, extracted from the `pola-rs/polars` project:[9]

Listing 2 Example of a PBT in the Test Oracle category

```
@given(value=strategy_datetime_ms)
def test_datetime_ms(value: datetime) -> None:
    result = pl.select(pl.lit(value, dtype=pl.
        Datetime("ms")))["literal"][0]
    expected_microsecond = value.microsecond //
        1000 * 1000
    assert result == value.replace(microsecond=
        expected_microsecond)
```

This test targets a function called `pl.Datetime("ms")`, which converts a datetime value to millisecond precision. Since the original datetime already includes microseconds, the test computes an expected value by truncating the microseconds to the nearest lower millisecond using integer division. The assert statement then verifies that the value returned

Table 2 Distribution of the categories in the 367 tests of the dataset

Categories	#	%
Test Oracle	110	29.97
Some Things Never Change	64	17.44
Hard to Prove, Easy to Verify	59	16.08
Roundtrip	51	13.90
Different Paths, Same Destination	24	6.54
Testing for Exceptions	18	4.90
Testing for Crashes	16	4.36
Outputs Within Expected Bounds	16	4.36
Idempotence	0	0.00
Other	9	2.45
Total	**367**	**100.00**

[9] https://github.com/pola-rs/polars/py-polars/tests/parametric/test_lit.py#L27

by the library `(result)` matches this expected value. Because the expected output is computed using a standard Python expression, the test was classified as a *Test Oracle*.

Next, in Listing 3, we present an example from the *Some Things Never Change* category, extracted from the `dbrattli/Expression` project.[10] This test checks whether the function `block.of_seq` preserves the length of the input list `xs`—that is, whether the length remains unchanged after the transformation.

Listing 3 Example of a PBT in the Some Things Never Change category

```
1 @given(st.lists(st.integers()))
2 def test_block_len(xs: List[int]):
3     ys = block.of_seq(xs)
4     assert len(xs) == len(ys)
```

Our third example, shown in Listing 4 and extracted from `google/caliban` project,[11] belongs to the *Hard to Prove, Easy to Verify* category. It verifies the behavior of a function that merges two dictionaries, `m1` and `m2`. First, it calls the `merge` function (line 3) to produce a new dictionary, `merged`. Then, it checks whether each key from `m2` has been correctly inserted into `merged` (lines 5–6). Finally, it ensures that all key-value pairs from `m1` are present in `merged`, unless a key was overwritten by `m2`.

Listing 4 Example of a PBT in the Hard to Prove Easy to Verify category

```
1  @given(st.dictionaries(st.text(), st.text()), st.
       dictionaries(st.text(), st.text()))
2  def test_merge(m1, m2):
3      merged = u.merge(m1, m2)
4      # Every item from the second map should be in
           the merged map.
5      for k, v in m2.items():
6          assert merged[k] == v
7
8      # Every item from the first map should be in
         the merged map, OR,
9      # if it shares a key with m2, m2's value will
           have bumped it.
10     for k, v in m1.items():
11         assert merged[k] == m2.get(k, v)
```

As can be seen in Table 2, 2.54% of the tests do not fit into any of the proposed categories, as they do not validate properties that can be generalized. Listing 5 shows an example of such a test, which generates random values for a case[12] object but checks a very specific condition—such as whether a value is equal to a given number.

[10] https://github.com/dbrattli/Expression/tests/test_block.py#L180

[11] https://github.com/google/caliban/tests/caliban/util/test_util.py#L120

[12] https://github.com/schemathesis/schemathesis/test/specs/openapi/test_examples.py#L806

Listing 5 Example of a PBT in the Other category

```
@given(case=strategy)
@settings(deadline=None, suppress_health_check=
    list(HealthCheck), phases=[Phase.generate])
def test(case):
    assert "\x00" not in case.query["q1"]["foo"]
    assert len(case.query["q2"]["foo"]) == 4
    assert int(case.query["q2"]["foo"]) % 2 == 0
```

Summary: After analyzing 367 PBTs, we found that the most common category by far is *Test Oracle* (29.97%), followed by *Some Things Never Change* (17.44%). The least common categories are *Testing for Crashes* (4.36%), *Outputs Within Expected Bounds* (4.36%) and *Idempotence* (0%). While this should not be the sole criterion for selecting which PBTs to implement first, we believe this information can be helpful for those beginning to adopt property-based testing.

3.4 What are the Most Commonly Used Built-In Constructs in Hypothesis? (RQ2)

As shown in Table 3, among the 367 PBTs, the most frequently used Hypothesis construct is *built-in strategies*—the input generation strategies provided by Hypothesis—appearing in 75.20% of the tests. The next most common construct is `@settings`, used to configure test parameters, as detailed below. By contrast, the least used construct is `note()`, which appears in only 0.82% of the tests. It is important to note that a single test may employ multiple constructs simultaneously, which explains why the total percentages in Table 3 exceed 100%.

3.4.1 Most Common Strategies

Since these are the most popular constructs, we further investigated the strategies used in the PBTs from our dataset. The results are presented in Table 4. As shown, the `integers()` strategy is the most frequently used, accounting for 31.60% of occurrences. Next is `sampled_from()`, which generates random values from an existing collection, with 16.04%, followed by `booleans()` at 14.39%. The analysis also revealed less common strategies, such as `dictionaries()` and `fixed_dictionaries()`, which generate dictionaries with fixed keys, together representing 2.59%, and `one_of()`, which selects values from multiple strategies, at 1.89%.

An interesting observation is that only 75.20% of the tests use *built-in strategies* (as mentioned in Table 3). To better understand this, we investigated the strategies employed in the remaining tests, since it would be unusual for a test not to use any strategy. Our analysis revealed three other types of strategies, which we classify as external (i.e., implemented by

Table 3 Frequency of built-in constructs

Constructs	#	%
built-in strategies	276	75.20
@settings	111	30.25
assume()	26	7.08
@example	19	5.18
note()	3	0.82

Table 4 Frequency of the built-in strategies

Strategies	#	%
integers()	134	31.60
sampled_from()	68	16.04
booleans()	61	14.39
lists()	40	9.43
text()	33	7.78
floats()	31	7.31
binary()	13	3.07
dictionaries() / fixed_dictionaries()	11	2.59
data()	9	2.12
one_of()	8	1.89
builds()	6	1.42
tuple()	6	1.42
from_regex()	4	0.94
Total	**424**	**100.00**

Table 5 Frequency of each type of strategy

Strategies	#	%
built-in strategies	276	75.20
external strategies	83	22.62
internal strategies	63	17.17
extension strategies	13	3.54

external projects), internal (i.e., implemented within the same project as the PBTs under analysis), and extension strategies (i.e., implemented at the `hypothesis.extra` module but to generate data for external libraries). Table 5 presents the frequency of each. As can be seen in this table, the categories are not disjoint and, therefore, the percentages add up to more than 100%. For example, we have 40 tests that use both built-in and external strategies.

As mentioned, *external strategies* are provided by modules or packages outside Hypothesis, including general-purpose libraries. An example is shown in Listing 6, which illustrates the `anything()` strategy from the project `pfun`, a Python library for functional programming.[13] This strategy generates arbitrary values of any type supported by the library, thereby providing the test with diverse input data. In the example, two instances of `anything()` are used for the first and second parameters (line 3), ensuring the evaluation of different value combinations.

Listing 6 Example of an external strategies

```
1  from pfun.hypothesis_strategies import anything,
       lists, maybes, unaries
2
3  @given(anything(), anything())
4   def _test_nothing_inequality(self, first, second
       ):
5           assume(first != second)
6           assert Just(first) != Just(second)
```

We also analyzed the projects responsible for implementing the external strategies, as shown in Table 6. The project that implements most external strategies was `Caffe2`, a deep

[13] https://github.com/suned/pfun/tests/test_maybe.py#L60

Table 6 Distribution of external strategies by project

Project	#	%
caffe2	35	42.17
janitor	7	8.43
static-frame	7	8.43
pfun	5	6.02
hypothesis-gufunc	5	6.02
ivy	4	4.82
PyTorch	4	4.82
brownie	3	3.61
electionguard-tools	3	3.61
other	10	12.05
Total	**83**	**100.00**

learning framework developed by Facebook, appearing in 42.17% of the tests that use external strategies. The next most frequent projects are `Janitor`, a utility library for data preprocessing in pandas and `Static-Frame`, an alternative to pandas, both accounting for 8.43% of the external strategies in our dataset.

Internal strategies refer to data generation strategies implemented within the project's codebase, either as internal modules or file-scope variables. An example is the `st_oid` function shown in Listing 7, which is part of the *gecko-dev* project.[14] This strategy generates object identifiers (OIDs), which are sequences of numbers, and ensures their validity by defining rules for the first two numbers and the total length of the sequence.

Listing 7 Example of an internal strategy to generate object identifiers (OIDs)

```
@st.composite
def st_oid(draw, max_value=2**512, max_size=50):
    """
    Hypothesis strategy that returns valid OBJECT
        IDENTIFIERs as tuples

    :param max_value: maximum value of any single
        sub-identifier
    :param max_size: maximum length of the
       generated OID
    """
    first = draw(st.integers(min_value=0,
       max_value=2))
    if first < 2:
        second = draw(st.integers(min_value=0,
           max_value=39))
    else:
        second = draw(st.integers(min_value=0,
           max_value=max_value))
    rest = draw(st.lists(st.integers(min_value=0,
        max_value=max_value),
                         max_size=max_size))
    return (first, second) + tuple(rest)
```

Finally, *extension strategies* refer to the use of the `hypothesis.extra` module, which provides strategies to extend Hypothesis with support for external libraries. While `hypothesis.strategies` offers generators for basic Python types, `hypothesis.`

[14] https://github.com/mozilla/gecko-dev/third_party/python/ecdsa/ecdsa/test_der.py#L379

`extra` focuses on data types from popular frameworks, such as NumPy[15] and Django[16], which are not part of the standard library. In other words, these strategies are implemented by the Hypothesis developers, similar to the built-in strategies. However, instead of generating Python types, they support tests requiring types from other libraries, as shown in Table 7. For this reason, we classified them as a separate category.

3.4.2 Most Common Settings

We also analyzed the usage of the `@settings` decorator, used to define test parameters or attributes, as summarized in Table 8. The `deadline` attribute, which sets the maximum execution time of a test, is the most common, appearing in 81.08% of tests. The `max_examples` attribute, which controls the maximum number of generated examples, follows at 48.65%. Less frequent attributes include `@suppress_health_check`, which suppresses performance warning messages (16.22%), and `verbosity`, which adjusts the level of detail in the log output (13.51%). Finally, the `parent` attribute, used to inherit the configuration from a parent test profile, is the least used, appearing in only 0.90% of cases.

After identifying the prevalence of the `deadline` and `max_examples` attributes, we conducted a detailed analysis of their configurations to understand how they are used in practice. As shown in Table 9, among the 90 tests that configure the `deadline` attribute, the vast majority (72.22%) assign it the value `None`. This choice effectively disables the execution deadline, indicating a trend toward tests without time constraints, where example generation can proceed indefinitely or until other limits are reached (such as the maximum number of examples). It is also worth noting that, in the absence of explicit configuration, Hypothesis applies a default `deadline` of 200 milliseconds. The second most common configuration sets the value to 10,000 milliseconds (16.67%), suggesting that when developers do enforce a time limit, it is typically much more permissive than the default.

Table 7 Frequency of extension strategies by extended library

Extended Library	#	%
hypothesis.extra (numpy)	11	84.62
hypothesis.extra (django)	1	7.69
hypothesis.provisional	1	7.69
Total	**13**	**100.00**

Table 8 Frequency of the @settings attributes

Attributes	#	%
deadline	90	81.08
max_examples	54	48.65
suppress_health_check	18	16.22
verbosity	15	13.51
parent	1	0.90

[15] https://github.com/numpy/numpy

[16] https://github.com/django/django

Table 9 Frequency of deadline values

deadline	#	%
None	65	72.22
1000	3	3.33
2000	2	2.22
2500	1	1.11
10000	15	16.67
20000	1	1.11
30000	1	1.11
datetime	2	2.22
Total	**90**	**100.00**

Table 10 Frequency of max_examples values

max_examples	#	%
2	3	5.26
3	3	5.26
5	3	5.26
10	21	36.84
20	8	14.04
50	3	5.26
external	2	3.51
other values	14	24.56
Total	**57**	**100.00**

For the `max_examples` attribute, which defines the maximum number of examples generated per test, the analysis revealed a distinct pattern compared to the default value set by Hypothesis (100 examples). As shown in Table 10, the most commonly requested number of examples is 10, appearing in 36.84% of tests. The second most common is 20 examples, found in 14.04% of the analyzed tests.

> **Summary:** Strategies are the most frequently used Hypothesis construct (75.20%), followed by `@settings`. We also identified three other types of strategies: external (22.62%), internal (17.17%), and extensions (3.54%). Regarding test parameters, the most common are `deadline` (81.09%), which sets the maximum execution time for tests, and `max_examples` (48.65%), which defines the maximum number of test cases automatically generated by Hypothesis.

3.5 Threats to Validity

In this first study, we used a sample of 367 PBTs, which prevents us from drawing more general claims. We also focused on only one language (Python) and only one testing framework (Hypothesis). However, the most relevant threat lies in the manual classification adopted to answer RQ1. We analyzed PBTs from real-world systems, which often involve complex logic. Therefore, our classification is subject to errors and questionable decisions. To limit such errors, the classification was carried out by two authors and complemented with two consensus meetings. In addition, our categories were, for the most part, extracted from documents widely referenced by PBT practitioners. On the other hand, counting the con-

structs used in the implementation of PBTs—as performed in order to answer RQ2—is an objective task and, for this reason, it was performed by only one author. Even so, since it was conducted manually, we consider it may be subject to small errors.

4 Stack Overflow Study

In this section, we analyze Stack Overflow questions to identify the key challenges developers encounter when using the Hypothesis framework. We aim to answer the following research question:

RQ3: *What are the primary challenges developers face when using the Hypothesis framework?* With this question, our goal is to leverage content from a forum like Stack Overflow to identify the main challenges developers face when using Hypothesis. Our aim is to help new PBT developers (by alerting them to common points of confusion) and also assist framework developers (providing them with consolidated information about the questions and challenges faced by users who have chosen to adopt this type of testing). However, it is important to note that our analysis aims to reveal the most frequent categories of questions. In particular, we did not set out to evaluate the complexity of a question (or of its answer), nor its severity (for example, the possibility of a test producing false positives).

4.1 Dataset and Study Design

The first author of this paper retrieved and analyzed key Stack Overflow questions related to the use of Hypothesis. After a few trial searches, she decided to include only questions tagged as *python-hypothesis* (209 questions) or *property-based-testing* (147 questions). However, among the 147 questions tagged as *property-based-testing*, 120 were considered false positives, as they concerned other testing frameworks or programming languages. Specifically, these 120 questions also had at least one of the following tags:

> *fscheck, quickcheck, scalacheck, jqwik, kotest, haskell, scala, c++, java, kotlin, typescript, javascript, rust, clojure, go, erlang, ocaml, f#, functional-programming*

Therefore, they were discarded. This left 236 questions: 209 with the *python-hypothesis* tag and 27 with the *property-based-testing* tag. However, 20 questions had both tags. Thus, in terms of unique questions, 216 remained. Among these unique questions, three were also discarded (one due to copyright infringement and two that, despite having the *python-hypothesis* tag, were not related to the topic). Consequently, the final dataset is composed of 213 questions.

As a second task, the first author conducted an analysis to identify patterns that would allow the questions to be grouped based on the nature of the problems reported by users, resulting in the inductive identification of seven distinct categories: *Strategies for Data Generation*, *Framework Features*, *General Questions*, *Optimization and Performance*, *Use of Decorators*, *Integration with External Libraries and Reporting*, and *Results Visualization*.

Then, the first and third authors independently performed the manual classification of the questions. Each question was carefully assigned to a category considering the context and the nature of the problem described. Finally, the authors met to discuss and resolve the

cases where there was disagreement in the classification. In such cases, the final decision was guided by the objective of the question, i.e., what the user was trying to clarify. The Kappa coefficient computed at the end of the individual classifications was 0.80, indicating a substantial level of agreement between the classifiers.

As we will report next, 78 questions (36.6%) were about data generation strategies. For this reason, the two authors decided to further detail this classification and also to re-analyze and classify the types of data developers intended to generate. Thus, independently, both authors classified the data generation strategies discussed in each of these 78 questions. They also held a second consensus meeting to reach the final classification. The Kappa coefficient computed at the end of the process was 0.93, indicating a very high level of agreement between the authors.

4.2 Challenges Developers Face When Using the Hypothesis Framework

Table 11 presents the final list of challenges identified in the 213 Stack Overflow questions related to the Hypothesis library. The category *Strategies for Data Generation* stood out prominently, accounting for 36.62% of the analyzed questions. This was followed by *Framework Features* (22.54%), *General Questions* (12.21%), and *Optimization and Performance* (9.39%). The categories with the fewest questions were *Use of Decorators* (7.51%), *Integration with External Libraries* (6.57%), and *Reporting and Results Visualization* (5.16%). Next, we discuss and provide examples of questions from each of these categories.

Strategies for Data Generation (78 questions): This category includes questions about custom data generation strategies or regarding the adaptation of existing strategies for specific tests. For example, in the question shown in Fig. 2, the developer asks whether Hypothesis can generate data for postal codes, phone numbers, and e-mail addresses.

Table 11 Distribution of the question categories in the 213 questions of the sample

Categories	#	%
Strategies for Data Generation	78	36.62
Framework Features	48	22.54
General Questions	26	12.21
Optimization and Performance	20	9.39
Use of Decorators	16	7.51
Integration with External Libraries	14	6.57
Reporting and Results Visualization	11	5.16
Total	**213**	**100.00**

Can hypothesis be used to generate data of a specific type (ie. zip codes or phone numbers)?

For example, I can define a test to generate integers, but I am expecting this test to conform to valid zip codes as well. Can I do this? Or, perhaps, more complex conform to non-US zip codes, which are all integers, but others (say, Canada) are not?

The same type of thing would be useful for text fields that are expected to conform to some type of mask (ie. email address).

Source: `https://stackoverflow.com/questions/44681295`

Fig. 2 Example question in the Strategies for Data Generation category

Python hypothesis: How to assert errors in test function

Simple example is a division function which works for integers >0:

(...) in this simple example I could simply exclude 0 because I know the error will happen. In other functions, especially the ones working with strings, it would be difficult to exclude all things where I know that an error is thrown. In many cases though the error is ok, e.g. when using null bytes in string functions. Is it possible to somehow 'allow' certain types of errors?

Source: https://stackoverflow.com/questions/68001322

Fig. 3 Example question in the Framework Features category

When to use Property-Based Testing?

I am trying to learn Property-Based Testing(PBT). I think I know how to implement it but when should I apply PBT? For example in this case I am trying to compare if the function getCurrentName() returns the expected name. Should I randomize this test?

```
@Test
public void getNameTest()
assertEquals(nameProxy, proxyFoto.getCurrentName());
```

Source: https://stackoverflow.com/questions/70333238

Fig. 4 Example question in the General Questions category

In Subsection 4.2.1, we provide a detailed analysis of the types of data developers attempt to generate with Hypothesis but face problems or unintended behavior.

Framework Features (48 questions): These questions are related to the behavior of Hypothesis and its features. Consider the question in Fig. 3, where the user seeks a way to "allow" certain errors in tests, such as those that only occur when processing specific strings.

General Questions (26 questions): These questions concern both comparisons between Hypothesis and other testing tools, as well as general, conceptual doubts about PBT. Consider the question in Fig. 4, where the user raises a broad issue regarding the application of PBT.

Optimization and Performance (20 questions): This category covers questions related to improving test performance, reducing execution time, and optimizing resource usage. Consider the question in Fig. 5, where the developer looks for a more efficient way to generate matrices with independent columns, avoiding the computational overhead of Hypothesis's `assume` function.

Use of Other Decorators (16 questions): This category includes questions related to decorators provided by Hypothesis other than `@given`, such as `@composite`, `@seed`, `@example`, and `others`. For example, in the question in Fig. 6 the user wants to understand how to use the `@seed` decorator to reproduce a specific test.

Integration with External Libraries (14 questions): This category concerns questions about integrating Hypothesis with other testing libraries, plugins, or tools. Consider the

Generate matrix with independent columns

I am trying to generate matrices with independent columns. At the moment I am using assume which works, but requires a lot of computation:

(code example)

Is there a better way to directly create a matrix X with independent columns?

Source: https://stackoverflow.com/questions/73424597

Fig. 5 Example question in the Optimization and Performance category

How to use @seed in hypothesis?

I'm trying to use @seed here: (URL from Hypothesis Documentation)

But when I include it before @given, I get the error NameError: name 'seed' is not defined at runtime.

Source: https://stackoverflow.com/questions/73424597

Fig. 6 Example question in the Use of Other Decorators category

pip install hypothesis(pandas) says hypothesis3.82.1 does not provide the extra 'pandas'

pip install hypothesis - django - seemed to work and hypothesis.extra has django but not pandas. Any idea what is going on with the pip install for pandas and numpy extras?

Source: https://stackoverflow.com/questions/53221061

Fig. 7 Example question in the Integration with External Libraries category

How to see the output of Python's hypothesis library

When using the hypothesis library and performing unit testing, how can I see what instances the library is trying on my code?

(code example)

The question is: how do I print / see the some_number variable, generated by the library?

Source: https://stackoverflow.com/questions/53072398

Fig. 8 Example question in the Reporting and Results Visualization category

question in Fig. 7, where the user wants to understand why installing support for Django succeeds, while installing support for Pandas fails.

Reporting and Results Visualization (11 questions): This category covers questions related to the visualization of data generated by Hypothesis during test execution, as illustrated in the question shown in Fig. 8.

4.2.1 Questions on Data Generation

Given the predominance of posts regarding *Strategies for Data Generation*, we chose to deepen our investigation to better understand the specific types of data users aim to generate. The results of this second classification are presented in Table 12. As shown, the *Composite Data* subcategory is the most frequent, accounting for 37.18% of all data generation questions. It is followed by *DataFrame and Tabular Data* (24.36%) and *Primitive Data* (19.23%).

Next, we describe these five subcategories of data generation posts:

Composite Data (29 questions): This subcategory refers to questions about generating composite structures or collections, such as tuples, dictionaries, lists, and arrays. Consider Fig. 9, where the user wants to generate dictionaries with values of different types (e.g., int, string, among others). Although `st.lists` provides the `unique_by` parameter to enforce variety within lists, `st.dictionaries` does not offer a similar mechanism. The core challenge, therefore, is finding a way to ensure that generated dictionaries contain values of distinct types.

DataFrame and Tabular Data (19 questions): This subcategory focuses on questions about generating structured data in tabular format, often using libraries such as pandas. Consider Fig. 10, where the user is generating a `pandas.DataFrame` containing float columns and a date index. The issue is that the generated columns frequently contain only NaN (*Not a Number*) values. Thus, the user needs to ensure two conditions: (1) No column should consist exclusively of NaN values; (2) The DataFrame must not be empty.

Primitive Data (15 questions): This subcategory covers questions about generating basic data types and their simple variations, such as integers, floats, text, booleans, and

Table 12 Distribution of the types used in the Strategies for Data Generation category

Subcategories	#	%
Composite Data	29	37.18
DataFrame and Tabular Data	19	24.36
Primitive Data	15	19.23
Object Data	8	10.26
Complex and Recursive Data	7	8.97
Total	**78**	**100.00**

Hypothesis search strategy for dictionaries with different types of values

I am trying to generate dictionaries containing different python types as values using the hypothesis module. For lists I can do this simply using the expression

(code example)

But for dictionaries, there is no `unique_by` for the values.

Is there an easy way to generate 'a': int, 'b': str, .. (at least two different python types in dict.values())?

Source: `https://stackoverflow.com/questions/59107181`

Fig. 9 Example question in the Composite Data subcategory

Python hypothesis dataframe assume column not exclusively consisting of NaN values

I am producing a pd.DataFrame using the hypothesis library like so:

(code example) + (table example)

I need to make sure that an individual column doesn't exclusively consist of NaN values. Also, I want to avoid to create an empty pd.DataFrame.

Source: https://stackoverflow.com/questions/78576317

Fig. 10 Example question in the DataFrame and Tabular Data subcategory

Hypothesis.strategies generate string from date

My code generates date as datetime.date object, but I need it in string format (01.01.2020). So I need to convert it like

```
random_date.strftime("%d.%m.%Y")
```

But I can't find any way to do that. Is it possible to generate string from date in hypothesis?

Source: https://stackoverflow.com/questions/66704166

Fig. 11 Example question in the Primitive Data subcategory

Randomizing a user dataclass with pytest and hypothesis

According to the docs, it seems like it should be possible to use an auto-generated address builder:

(code example)

But when I try the following options, neither work:

```
st.register_type_strategy(Address)
@given(Address)
```

Source: https://stackoverflow.com/questions/73770201

Fig. 12 Example question in the Object Data subcategory

dates. Consider Fig. 11, where the user is generating random dates with `strategies.dates()`, but needs the result returned as a string in the format `"dd.mm.yyyy"` instead of a `datetime` object.

Object Data (8 questions): This subcategory involves generating objects with specific attributes and behaviors. Consider the question in Fig. 12, where the user is able to manually create instances of the `Address` class but expects Hypothesis to automatically construct such objects from annotated types (e.g., `street: str`).

Complex and Recursive Data Structures (7 questions): This subcategory covers the generation of hierarchical and recursive structures, such as trees and nested lists. Consider the

How can I generate a boolean expression recursively in Python Hypothesis?

I am new to Python's Hypothesis library and property based testing in general. I want to generate arbitrarily nested policy expressions with the following grammar: ((A and B) or C)

I am feeling that the recursive strategy is what I want, but I'm having a hard time understanding how to use it. The code I have only seems to generate one "level" of expression.

(code example)

How might I generate arbitrarily nested policy expressions using Hypothesis?

Source: `https://stackoverflow.com/questions/52207646`

Fig. 13 Example question in the Complex and Recursive Data Structures subcategory

question in Fig. 13, where the user wants to generate boolean expressions following the structure `((A and B) or C)`. They believe that a recursive function could help, but their current implementation produces only single-level combined expressions.

> **Summary:** The analysis of the 213 questions revealed that the main difficulties developers face are related to *Strategies for Data Generation*, which accounts for 36.62% of our dataset. Within this category, the subcategory *Composite Data* is the primary source of questions, representing 37.18% of the data generation posts.

4.3 User Reputation Analysis

In Stack Overflow, users are awarded reputation points for posting or answering questions, getting votes on their posts, among other tasks. Therefore, reputation is a common metric for assessing a user's activity, and a high reputation tends to indicate expertise and trust within the Stack Overflow community (Bosu et al. 2013). Thus, we decided to investigate the correlation between the categories of questions and the reputation of the Stack Overflow users who asked those questions.

Among the 213 questions, there were 160 unique users. However, two questions were excluded due to unavailable reputation information (i.e., deleted user accounts). Table 13 summarizes the reputation distribution of the remaining users.

Table 14 shows the question category, the number of questions in that category, then the number of questions by users in the first (Q1), second (Q2), third (Q3), and fourth quartiles (Q4), in terms of the distribution of their reputation scores.

Table 13 Reputation Distribution Summary

Minimum (0th percentile)	1
Q1 (25th percentile)	290
Q2 / Median (50th percentile)	1,332
Q3 (75th percentile)	5,177
Maximum (100th percentile)	475,933

Table 14 Distribution of the question categories according to the user reputation quartile

	Total	User Reputation			
Categories	Questions	Q1	Q2	Q3	Q4
Strategies for Data Generation	77	23	16	23	15
Framework Features	48	7	13	15	13
General Questions	25	10	4	6	5
Optimization and Performance	20	0	9	8	3
Use of Decorators	16	6	3	3	4
Integration with External Libraries	14	4	3	3	4
Reporting and Results Visualization	11	3	4	2	2
Total	**211**	**53**	**52**	**60**	**46**

For *Strategies for Data Generation*, more questions (23/77 ≈ 30%) were asked by users in the first and third quartiles. Proportionally, *General Questions* was the category with the highest share of users in the first quartile (10/25 ≈ 40%). On the other hand, there were no questions about *Optimization and Performance* asked by users in the first quartile. Finally, there was no category in which users from the fourth quartile asked more questions than users from the other quartiles.

We also performed a chi-square test to assess the association between the number of answers and reputation quartiles. To address the chi-square assumptions, we merged the first two quartiles (Q1 and Q2) and the last two quartiles (Q3 and Q4), avoiding cells with low expected frequencies. This binary partitioning allows us to compare lower- and higher-reputation users. The results indicate no statistically significant association between the categories and the reputation quartiles (Q1+Q2 vs. Q3+Q4), with similar category distributions across the two groups (p-value = 0.813).

4.4 Threats to Validity

Similar to the first study, the classification of Stack Overflow posts is subject to errors and debatable decisions. To mitigate such risks, the classification was performed by two authors. The categories, however, were proposed solely by the first author. The second author, nonetheless, consistently concurred with this proposal and did not identify the need to introduce additional categories. It is worth emphasizing that this classification of Stack Overflow posts was simpler and more objective than that conducted in the first study. This observation is supported by the Kappa coefficients, which were 0.54 (RQ1) and 0.80 (RQ3).

It is also important to note that, in this study, after a data-cleaning step, we analyzed 213 questions, which represent the valid questions about PBTs implemented using Hypothesis available on Stack Overflow. However, we cannot generalize our results as representative of the concerns of all developers using PBT, since the study was restricted to the Stack Overflow environment.

5 Assessing a Tool to Generate PBTs

In this third study, we use Ghostwriter (for automatic test generation, as described in Subsection 2.3 and Appendix A) to answer the following research question:

RQ4: *Is the Ghostwriter tool capable of generating PBTs effectively?* Our goal is to verify whether developers who already use, or intend to use, PBTs can rely on Ghostwriter

to automate test generation. To answer this question, we build on the results of RQ1 and organize our analysis by PBT categories.

5.1 Study Design

Of the nine categories proposed in Subsection 3.1, Ghostwriter is only capable of generating tests for four categories from our dataset: *Roundtrip*, *Test Oracle*, *Different Paths, Same Destination*, and *Testing for Exceptions*. Therefore, these are the only categories analyzed and considered when answering RQ4.

Initially, the first author carefully evaluated the 203 tests from these four categories, which accounted for 55.31% of the 367-test dataset used to answer RQ1. The remaining tests were excluded because they belong to categories not supported by Ghostwriter. In this evaluation, the author essentially assessed whether the implementation of the selected tests was compatible with, and followed the same structure as the code skeletons generated by Ghostwriter, as detailed in Subsection 2.3 and also in Appendix A.

The tests were marked as follows:

- *Eligible for automatic generation:* a test is considered eligible when it fully matches the code skeleton assumed by Ghostwriter.
- *Partially eligible for automatic generation:* a test is considered partially eligible when it matches the code skeleton assumed by Ghostwriter but also includes additional commands, thus requiring adjustments and improvements.
- *Ineligible for automatic generation:* a test is considered ineligible when it definitively does not match the code skeleton assumed by Ghostwriter.

5.2 Results

As shown in Table 15, 105 tests (51.72%) were classified as *ineligible* for automatic generation. This highlights that template-based tools like Ghostwriter faces significant limitations when used to support the generation of tests in real-world scenarios, such as those in our dataset. In contrast, 37 tests (18.23%) were classified as *eligible*, meaning they could be successfully generated by Ghostwriter. Finally, 61 tests (30.05%) were classified as *partially eligible*.

Next, we present three examples of tests from the *Roundtrip* category, illustrating different generation outcomes. Listing 8 shows a test classified as eligible for automatic generation by Ghostwriter.[17]

Table 15 Eligibility distribution across categories

Category	Eligible	Partially Eligible	Ineligible
Test Oracle	15	26	69
Roundtrip	12	22	17
Different Paths, Same Destination	9	9	6
Testing for Exceptions	1	4	13
Total	**37**	**61**	**105**

[17] https://github.com/mozillazg/pypy/rpython/rlib/test/test_rbigint.py#L1676

Listing 8 Example of an Eligible test in the Roundtrip category

```
1  @given(longs)
2  def test_unsigned_roundtrip(self, x):
3      x = abs(x)
4      rx = r_uint(x) # will wrap on overflow
5      assert rbigint.fromrarith_int(rx).touint() ==
           rx
6      rx = r_ulonglong(x) # will wrap on overflow
7      assert rbigint.fromrarith_int(rx).toulonglong
           () == rx
```

When comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function (presented in Subsection 2.3), we can see that its structure exactly matches what Ghostwriter generates. Therefore, we conclude that this test could have been generated by Ghostwriter.

On the other hand, Listing 9 presents an example of a test classified as partially eligible.[18]

Listing 9 Example of an Partially Eligible test in the Roundtrip category

```
1  @given(x=strategies.input_map["double"])
2  def test_double_inverse(x):
3      encoded = fixed.encode_double(x)
4      decoded, pos = fixed.decode_double(encoded,
           0)
5      assert pos == len(encoded)
6      if math.isnan(x):
7          assert math.isnan(decoded)
8      else:
9          assert decoded == x
```

Comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function, we can see that the essential structure is present. For example, the test declares variables whose values are obtained from different methods (lines 3–4), and then an assertion compares both values. However, the test also includes additional commands, such as an `if/else` statement and a second assertion that checks the `math.isnan` function (lines 5–9), which are not part of the skeleton generated by Ghostwriter. Thus, we consider it to be only partially generable by Ghostwriter.

Finally, the test in Listing 10 illustrates an example of a test not eligible for automatic generation by Ghostwriter.[19]

[18] https://github.com/nccgroup/blackboxprotobuf/lib/tests/py_test/test_fixed.py#L135

[19] https://github.com/cdown/srt/tests/test_srt.py#L305

Listing 10 Example of an Ineligible test in the Roundtrip category

```
@given(st.lists(subtitles()))
def test_parsing_content_with_blank_lines(subs):
    for subtitle in subs:
        # We stuff a blank line in the middle so as to trigger the "special"
        # content parsing for erroneous SRT files that have blank lines.
        subtitle.content = subtitle.content + "\n\n" + subtitle.content

    reparsed_subtitles = srt.parse(srt.compose(subs, reindex=False, strict=False))
    subs_eq(reparsed_subtitles, subs)
```

When comparing this test with the skeleton generated by Ghostwriter's `roundtrip()` function, we notice that the typical skeleton structure is missing. For example, although the skeleton requires an assertion to verify the expected behavior, no explicit assertion appears in the code. Instead, this check is encapsulated within the `subs_eq` function. This distinction leads us to conclude that this test could not have been generated by Ghostwriter.

> **Summary:** The analysis of 203 tests revealed that only 18.23% could have been automatically generated by Ghostwriter, while 30.05% were classified as *Partially Eligible* and 51.72% as *Ineligible*. These results highlight significant limitations in Ghostwriter's ability to generate tests in practical scenarios. However, performance varied across categories: *Different Paths*, *Same Destination* showed the highest success rate (37.5%), whereas *Test Oracle* and *Testing for Exceptions* had the highest proportions of ineligible tests with 62.73% and 72.22%, respectively.

5.3 Threats to Validity

The classification of tests as eligible, partially eligible, or ineligible was performed manually by a single author, which may introduce subjectivity. However, the author always tried to base the decision on the template generated by the Ghostwriter tool. If the test followed the template, it could be classified as eligible (only minor changes to the template) or partially eligible (significant changes to the template, including the addition of statements). Otherwise, that is, if the template could not be identified in the code, the result of the classification was ineligible. Therefore, we believe that this criterion, which was carefully applied to all 203 tests, helps to reduce the chances of misclassification.

We would also like to mention that there may exist simpler tests that can be generated by Ghostwriter but were not implemented by the developers of the analyzed projects (for example, due to lack of time). If this is the case, these tests will function as false negatives in our study.

6 Lessons Learned and Recommendations

The systematic analysis of PBTs in open-source projects and the investigation of developer discussions in Stack Overflow reveal several opportunities for developers, tool maintainers, and researchers. For example, start adopting PBTs effectively, we recommend beginning with the most frequent categories. However, although *Test Oracle* is the most popular category (29.97%), it tends to have tests that are more complex and that have higher LOC (median = 21). Thus, a more accessible starting point might be the *Roundtrip* category, which is the fourth most frequent (13.90%) and can also be easily automated by Ghostwriter (23.53%). Another option is the *Different Paths, Same Destination* category, which, although less popular (6.54%), presents tests with lower LOC (median = 8) and a higher success rate in automatic generation (37.50%). In summary, prioritizing categories with smaller and popular tests might help to master PBT concepts before advancing to more relevant challenges.

The majority of developers' questions on Stack Overflow are related to data generation strategies (36.62%). This aligns directly with the fact that strategies are the most used constructor of Hypothesis in real projects. Specifically, generating more complex structures, such as dictionaries and lists (i.e., *Composite Data*), presents a significant challenge. Thus, resources like the official Hypothesis documentation and practical examples found on Stack Overflow are invaluable for building this knowledge base.

The study also highlights areas for improvement in the Hypothesis framework and its documentation. As mentioned before, the difficulty in generating *Composite Data*, evidenced by the large number of questions on Stack Overflow about the topic (37.18%), suggests the need for more detailed examples and guides. In addition, the recurrence of questions about *Optimization and Performance* (9.39%) indicates an opportunity to offer more guidance and tools that help developers write more efficient PBTs.

The evaluation of the Ghostwriter tool suggests that the automatic generation of PBTs is more effective for categories like *Different Paths, Same Destination* (37.50%) of success rate. However, more complex tests, such as those in the *Test Oracle* category, require significant intervention from developers. Future research could focus on developing techniques to automatically generate tests for all categories, including the simplest ones (whose success rate can still be improved) and, of course, for the more complex ones. Finally, it is important to mention that Ghostwriter currently does not generate tests for at least four categories (*Some Things Never Change*, *Hard to Prove, Easy to Verify*, *Outputs Within Expected Bounds*, and *Testing for Crashes*).

However, overall, the results obtained with Ghostwriter were not very promising. For instance, in our sample, even after removing tests from categories that are not supported by Ghostwriter, only 18.23% could be generated by the tool. Therefore, we believe that these results may motivate research on new tools for PBT generation that are not based on templates, such as tools based on AI models, including Large Language Models (LLMs). These tools generally show very promising performance for generating traditional tests, such as unit tests (Schäfer et al. 2024), and it may therefore be interesting to also evaluate their use for generating PBTs.

7 Related Work

We organized the discussion of related work into five parts: studies using functional languages; studies using Python (in generic domains); studies using Python with a focus on machine learning projects; studies of automatic test generation; and other studies.

7.1 Studies of PBT in Functional Languages

Arts et al. (2006) present a case study on the usage of PBT in an industrial implementation of the Megaco protocol (a pattern for controlling media gateways in telecommunications networks). In this case, the tests were implemented using the Quviq QuickCheck tool, which is a property-based testing tool for Erlang.[20] The article describes the input generators used by the tests, the features that are tested, and the methodology employed in the tests. The authors also commented that PBT was able to find significant failures that required non-trivial fixes. However, this study is based on a single application, while in our study, we analyzed the PBTs implemented in 244 well-known Python projects.

Goldstein et al. (2024) conduct a qualitative study in an industrial context at the financial technology company Jane Street, investigating the use of PBT through 30 interviews with developers who use the QuickCheck tool. The study identifies four recurring types of properties employed in practice, including the *Roundtrip* property, which is also analyzed in our study. About one-third of the participants reported that PBTs uncovered bugs not detected by other testing methods, reinforcing their potential to identify subtle faults in complex code. Despite these benefits, the authors highlight important challenges, such as the difficulty of deciding which properties to test and the cognitive cost of writing specifications, which leads many developers to adopt PBT opportunistically, restricting its use to properties considered obvious. In contrast, our study covers a broad set of property categories, offering support to help developers identify the properties to be tested.

7.2 Studies of PBT in Python

An empirical study evaluates the use and effectiveness of PBT in Python using Hypothesis. Conducted by Ravi and Coblenz (2025), this study proposes a taxonomy consisting of twelve property categories and analyzes their fault-detection capabilities through mutation testing. Their categorization is based on the assertions present in each test function, whereas our study performs categorization at the test level; however, the authors observed that a test function contains, on average, 1.65 distinct categories, indicating that PBT tests rarely combine multiple property types within a single function. They also identified recurring patterns, such as tests focused on exception raising, a pattern that is likewise observed in our dataset. Although their study provides important insights into the effectiveness of different PBT categories, it primarily focuses on evaluating error-detection performance. In contrast, our work investigate developers' usage patterns, practical challenges, and tool-support limitations, taking a broader perspective on the practical use of PBT in real-world projects. Moreover, our results complement existing findings by confirming similar trends, such as the prevalence of Roundtrip properties (13.90% in our study versus 13.84% in theirs), while

[20] https://www.quviq.com/products/

also providing new evidence on practices, adoption, and tool-support limitations that were not examined in prior work.

Hatfield-Dodds (2020) proposed four categories of properties relevant to property-based testing in scientific software, demonstrating how categories like *Outputs Within Expected Bounds*, *Roundtrip*, *Differential Testing*, and *Metamorphic* can help reveal bugs in well-known libraries such as NumPy and Astropy. While their work illustrates the implementation of these categories through specific examples, it does not quantify the practical occurrence of each. In our study, we adopted two of these categories, *Outputs Within Expected Bounds* and *Roundtrip*, to classify the tests in our dataset. Moreover, our study quantifies the occurrence of eight categories (as presented in Subsection 2.2) in a dataset of 367 PBTs implemented in open-source projects.

7.3 Studies of PBT in ML Projects

Wauters and De Roover (2024) focus on PBTs in ML projects, whereas our study analyzes individual PBTs across diverse domains, including web systems, libraries, and general-purpose software. Despite this difference, both studies highlight the importance of automatic data generation: Wauters et al. report positive developer perceptions, and our Stack Overflow analysis shows that most questions concern data generation in Hypothesis. They identify complex strategies as dominant, while our characterization reveals that integers and booleans are among the most commonly used. Additionally, `hypothesis.extra` (NumPy) is the most frequently used extension in our dataset. Finally, both studies note that PBTs are often applied to conventional code rather than ML-specific components.

In the study by Durelli et al. (2024), PBT is applied to test data generation for machine learning models. The authors propose a method that uses decision trees to intelligently create test data. This allows them to explore how the machine learning model behaves in various situations, helping to find errors that common tests might not detect.

7.4 Studies of Automatic Test Generation

Several studies have investigated software testing practices through large-scale empirical evaluations of automated testing, based on controlled experiments and quantitative metrics. Kracht et al. (2014) analyzed automatically generated and manually written test suites from 100 open-source Java projects, assessing test quality using branch coverage and mutation scores. Their study followed a standardized experimental protocol, with multiple executions under fixed computational budgets and statistical analyses to compare different tools. Similarly, Shamshiri et al. (2015) conducted a large-scale evaluation of automated unit test generation using 357 real faults from the Defects4J dataset. They measured test effectiveness in terms of fault detection and applied repeated executions and statistical tests to ensure result reliability. While these studies rely primarily on automated metrics and controlled experimental settings, our work adopts a difference empirical perspective. In contrast, we combine manual classification by independent evaluators, analysis of developer discussions, and tool evaluation to provide a more comprehensive view of testing practices in real-world settings.

Etemadi et al. (2025) propose an LLM-based approach for the automatic generation of PBTs aimed at protecting cyber-physical systems (CPS). The work introduces the ChekProp tool, which generates PBTs at design time from documentation, Python source code, and

unit tests, and reuses them at runtime as mechanisms for monitoring unsafe states. The tests are implemented using Hypothesis and follow a structure composed of generators, a test body, and assertions. Although this structure is similar to that of Ghostwriter, ChekProp is specific to CPS and relies heavily on existing artifacts, whereas Ghostwriter is a general-purpose tool applicable across different domains. The authors report a high recovery of properties compared to those defined manually, indicating the potential of LLMs to reduce human effort in critical domains.

Aichernig and Havelund (2023) investigate the use of LLMs to generate code and PBTs in Scala based on an Event-B–inspired model, in the context of a bridge traffic control system. Unlike more guided approaches, the authors delegate the near-complete generation of code and tests to ChatGPT, exploring two modeling strategies. The results reveal significant difficulties, such as faulty properties, violations of preconditions, and substantial effort required to understand and correct the generated code after multiple iterations. Taken together, these studies show that while LLMs are promising for supporting PBT generation, their effectiveness depends on the domain, the level of guidance provided, and the complexity of the model, which motivates our analysis of Ghostwriter's use in real-world Python projects.

7.5 Other Studies

Sun et al. (2024) use PBT to validate user privacy functionalities in social media. Through the PDTDroid tool, privacy specifications are converted into formal properties, enabling automatic detection of inconsistencies, such as data leaks. Meanwhile, Xiong et al. (2024) employ PBT to detect functional bugs that do not necessarily result in application crashes in Android apps. An example is when, upon removing a tag from OmniNotes, another tag in the same note or in the tag list is incorrectly affected To address this, the authors developed the Kea tool, which combines a property description language with automated exploration strategies to verify the expected behavior of applications.

8 Conclusion

This paper aimed to provide empirical insights into the practical application of PBT through the analysis of tests from well-known GitHub projects, developer questions, and support tools. By examining a random sample of 367 tests, the results showed that the most common category is *Test Oracle*, followed by *Some Things Never Change* and *Hard to Prove, Easy to Verify*. We also identified two new categories not previously addressed: *Testing for Exceptions* and *Testing for Crashes*. The tests analyzed were generally short, with a median of 14 LOC. Moreover, we found that strategies are the most frequently used Hypothesis constructs, especially `integers`, `lists`, and `booleans`. Our second study focused on the analysis of 213 Stack Overflow questions on PBT. This study revealed that the greatest challenges are related to data generation (36.6%), particularly composite structures (e.g., lists, arrays, tuples) and tabular data (e.g., dataframes). Other important challenges uncovered by the questions were related to framework features (22.54%). Finally, in a third study, we investigated the capabilities of the Ghostwriter tool in generating PBT. The tool has its limitations, and based on our experiments, it would only be able to generate 18% of the

tests automatically. However, about 30% more tests could be partially generated. Some test categories are better suited for auto-generation than others; for instance, *Different Paths, Same Destination* showed 37.5% of successfully auto-generated tests and another 37.5% of partially generated ones.

Based on the results of our three studies, we recommend that newcomers begin writing PBTs in the categories of *Different Paths, Same Destination* and *Roundtrip*, as both are popular categories and have a higher success for auto-generation using Ghostwriter. Particularly *Different Paths, Same Destination*, which has a low LOC median, and thus, is usually composed of smaller tests.

As future work, we suggest evaluating other widely adopted PBT frameworks beyond Hypothesis, such as `QuickCheck` for Haskell and Erlang and `jqwik` for Java. This would allow for comparative analyses across programming languages and software domains, helping to validate the generality of the categories and challenges identified in this study. We also propose investigating new solutions for the automatic generation of PBTs. A promising direction is the use of LLMs, exploring techniques capable of overcoming the challenges observed with Ghostwriter by learning from large code corpora to generate complex and realistic test cases. Another relevant direction is to broaden the analysis to additional forums and developer communities to capture diverse needs and practices, and to explore the application of PBT in different domains, such as machine learning models and Android applications.

A Ghostwriter-Generated Code Templates

In Section 2.3, we showed the code template generated by the Ghostwriter tool for roundtrip tests. In this section, we extend that presentation by showing the templates generated for the other PBT categories supported by Ghostwriter. Knowledge of these templates is important for a deeper understanding of the results of RQ4 (*Is the Ghostwriter tool capable of generating PBTs effectively?*).

Test Oracle: One way to generate tests for this category is through the `equivalent()` function. Suppose a program with two sorting functions: `quicksort` and `my_sort`. The first is a well-known, widely tested implementation, while the second is an alternative version that needs verification. Thus, quicksort[21] serves as an oracle for testing `my_sort`. Specifically, the following Ghostwriter call:

`hypothesis.extra.Ghostwriter.equivalent(quicksort, my_sort)`

generates the test:

```
@given(lst=st.nothing())
def test_equivalent_quicksort_my_sort(lst):
    result_quicksort = quicksort(lst)
    result_my_sort = my_sort(lst)
    assert result_quicksort == result_my_sort
```

[21] https://www.geeksforgeeks.org/dsa/python-program-for-quicksort/

It is noteworthy that `st.nothing()` is a placeholder generated by Ghostwriter. It should be replaced with a valid strategy like `st.lists(st.integers())` in practical usage.

Different Paths, Same Destination: When using Ghostwriter, one way to generate tests for this category is through the `binary_operation()` function. Suppose a function `add(a, b)` that takes two numbers and returns their sum. Then, we want to verify whether our implementation satisfies the associative property of addition. By calling:

`hypothesis.extra.Ghostwriter.binary_operation(add, associative=True, commutative=False,) identity=None)`

Ghostwriter generates the following test:

```
@given(a=st.nothing(), b=st.nothing(), c=st.
    nothing())
def test_associative_binary_operation_add(a, b, c
    ):
     left = add(a, add(b, c))
     right = add(add(a, b), c)
     assert left == right
```

By defining `associative=True`, we instruct Ghostwriter to generate tests for the associative property of the `add` function. Similarly, `commutative=False` and `identity=None` exclude tests for the commutative and identity properties.

Idempotence: When using Ghostwriter, one way to generate tests for this category is with the `idempotent()` function. Suppose we want to test a function `normalize()`, which takes a string, removes all whitespace, and converts all letters to lowercase. Then, by calling:

`hypothesis.extra.Ghostwriter.idempotent(normalize)`

Ghostwriter generates the following test:

```
@given(data=st.nothing())
def test_idempotent_normalize(data):
     result = normalize(data)
     repeat = normalize(result)
     assert result == repeat
```

Testing for Exceptions: When using Ghostwriter, one way to generate tests for this category is through the `fuzz()` function. Suppose we want to test whether a function `divide(x, y)` raises a `ValueError` when `y` is zero. By calling:

`hypothesis.extra.Ghostwriter.fuzz(divide, except_=(ValueError,)).`

Ghostwriter generates the following test:

```
@given(x=st.nothing(), y=st.nothing())
def test_fuzz_divide(x, y):
     try:
          divide(x, y)
     except ValueError:
          reject()
```

Finally, it is worth noting that the Ghostwriter does not support generating tests for the following categories: *Outputs Within Expected Bounds*; *Some Things Never Change*; *Hard to Prove, Easy to Verify*; and *Testing for Crashes*.

B Size Characterization

After obtaining the results of RQ1, we conducted an exploration of the sizes of the PBTs in our dataset, measured in lines of code (LOC). The idea was to provide a first quantitative characterization of our results. Figure 14 shows box plots of the resulting distributions.

When analyzed collectively (*All*), the median test size is 14 LOC. In other words, PBTs are generally small in terms of lines of code. The *Other* category has the highest median, at 30 LOC, while *Different Paths, Same Destination* has the lowest, at 8 LOC. Moreover, some categories include tests with widely varying sizes. For example, the *Test Oracle* category has a mean of 34.08 LOC and a standard deviation of 38.60 LOC. In fact, the mean is greater than the median in all of the categories analyzed, suggesting a right-skewed distribution.

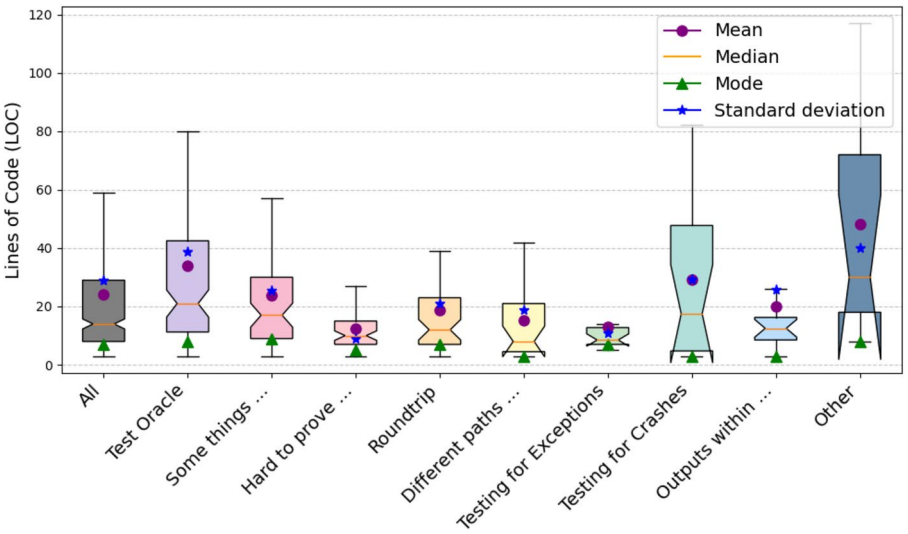

Fig. 14 Distribution of code lines for all categories

Author Contributions Isadora de Oliveira: Conceptualization, investigation, methodology, data analysis, validation, writing. Arthur Lisboa Corgozinho: Conceptualization, investigation, methodology, data analysis, validation, writing. Henrique Rocha: Conceptualization, investigation, methodology, validation, writing, supervision. Marco Tulio Valente: Conceptualization, investigation, methodology, validation, writing, supervision.

Funding The Article Processing Charge (APC) for the publication of this research was funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) (ROR identifier: 00x0ma614). FAPEMIG (process APQ-02419-23). CNPq (process 403304/2025-3).

Data Availability We publicly provide the complete dataset and replication package at https://doi.org/10.5281/zenodo.17117043, including spreadsheets for Study 1 (Characterization), Study 2 (Stack Overflow), and Study 3 (Ghostwriter Assessment).

Declarations

Conflict of Interest The authors declare that they have no conflict of interest. They also declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Ethical Approval Not applicable.

Informed Consent The authors consent to the use of this work in the journal.

Clinical Trial Number Not applicable.

References

Abdi M, Rocha H, Demeyer S (2022) Small-Amp: Test amplification in a dynamically typed language. Empirical Software Engineering (EMSE) 27(128)

Aichernig BK, Havelund K (2023) AI-assisted programming with test-based refinement. In: International Conference on Bridging the Gap between AI and Reality (AISoLA), pp 385–411

Arts T, Hughes J, Johansson J, Wiger U (2006) Testing telecoms software with Quviq QuickCheck. In: ACM SIGPLAN Workshop on Erlang, pp 2–10

Bosu A, Corley CS, Heaton D, Chatterji D, Carver JC, Kraft NA (2013) Building reputation in StackOverflow: An empirical investigation. In: 10th Working Conference on Mining Software Repositories (MSR), pp 89–92

Chen Z, Rizkallah C, O'Connor L, Susarla P, Klein G, Heiser G, Keller G (2022) Property-based testing: Climbing the stairway to verification. In: 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE), pp 84–97

Claessen K, Hughes J (2000) QuickCheck: a lightweight tool for random testing of Haskell programs. In: 5th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp 268–279

Corgozinho AL, Valente MT, Rocha H (2023) How developers implement property-based tests. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 380–384

Coutinho JCS, Andrade WL, Machado P (2023) Implementing exploratory testing in an agile context: a study based on design science research. In: 8th Brazilian Symposium on Systematic and Automated Software Testing (SAST), pp 67–76

Durelli VH, Monteiro R, Durelli RS, Endo AT, Ferrari FC, Souza SR (2024) Property-based testing for machine learning models. In: Symposium on Systematic and Automated Software Testing (SAST), pp 39–48

Etemadi K, Sirjani M, Helali Moghadam M, Strandberg P, Pettersson P (2025) LLM-based property-based test generation for guardrailing cyber-physical systems. In: International Conference on Bridging the Gap between AI and Reality (AISoLA), pp 18–46

Fink G, Bishop M (1997) Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes (SEN) 22(4):74–80

Fleiss JL, Cohen J (1973) The Equivalence of Weighted Kappa and the Intraclass Correlation Coefficient as Measures of Reliability. Educational and Psychological Measurement (EPM) 33(3):613–619

Goldstein H, Cutler JW, Dickstein D, Pierce BC, Head A (2024) Property-based testing in practice. In: 46th International Conference on Software Engineering (ICSE), pp 1–13

Hatfield-Dodds Z (2020) Falsify your Software: validating scientific code with property-based testing. In: 19th python in science conference, pp 162–165

Kracht JS, Petrovic JZ, Walcott-Justice KR (2014) Empirically evaluating the quality of automatically generated and manually written test suites. In: 14th International Conference on Quality Software (SWQD), pp 256–265

Löscher A, Sagonas K (2017) Targeted property-based testing. In: 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp 46–56

MacIver DR, Hatfield-Dodds Z et al (2019) Hypothesis: A new approach to property-based testing. J Open Source Softw 4(43):1891

Ravi S, Coblenz M (2025) An empirical evaluation of property-based testing in python. In: Proceedings of the ACM on Programming Languages (PACMLP) 9(OOPSLA2):3897–3923

Schäfer M, Nadi S, Eghbali A, Tip F (2024) An empirical evaluation of using large language models for automated unit test generation. IEEE Trans Software Eng 50(1):85–105

Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 201–211

Sun J, Su T, Sun J, Li J, Wang M, Pu G (2024) Property-based testing for validating user privacy-related functionalities in social media apps. In: 32nd ACM International Conference on the Foundations of Software Engineering (FSE), pp 440–451

Valente MT (2024) Software Engineering: A Modern Approach. Self-published

Vercacmmen S, Borg M, Demeyer S (2023) Validation of mutation testing in the safety critical industry through a pilot study. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 334–343

Wauters C, De Roover C (2024) Property-based testing within ML projects: an empirical study. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 648–653

Winters T, Manshreck T, Wright H (2020) Software engineering at Google: Lessons learned from programming over time. O'Reilly Media, Inc

Xiong Y, Su T, Wang J, Sun J, Pu G, Su Z (2024) general and practical property-based testing for android apps. In: 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 53–64