LLM-Pyro: LLM-Based Mutation Testing with Equivalence Triage for Python

Victor Manuel Chang
Federal University of Mato Grosso do Sul, UFMS
Campo Grande, Brazil
victor.manuel@ufms.br

Marcelo Manoel de Lima Filho
Federal University of Mato Grosso do Sul, UFMS
Campo Grande, Brazil
marcelo_l_filho@ufms.br

Awdren de Lima Fontão
Federal University of Mato Grosso do Sul, UFMS
Campo Grande, Brazil
awdren.fontao@ufms.br

Abstract

Mutation testing assesses test suite quality by injecting controlled faults into production code. Recent studies show that large language models (LLMs) can generate more diverse mutants, closer to real faults, than traditional syntactic operators, though at the cost of more non-compilable, duplicate, and equivalent mutants. This evidence, however, is concentrated in Java, leaving Python's dynamic typing largely unexplored, and existing tools either rely on fixed operators or apply LLMs without addressing equivalent mutants. We present LLM-Pyro, a Python adaptation of LLMorpheus that couples mutant generation with an LLM-based oracle classifying survivors as equivalent, non-equivalent, or unknown. We evaluate LLM-Pyro in a pilot study on HumanEval against mutmut and Mutahunter, and on real faults from PyBugHive. On HumanEval, LLM-Pyro produced more syntactically diverse mutants (average edit distance 2.96), with 72.2% compilability, 15.3% duplication, and 10.4% equivalence, generating 1.85× more mutants than mutmut and 4.53× more than Mutahunter (58.4% non-equivalent). On PyBugHive, within the 44 evaluated bugs from four projects, LLM-Pyro reached an 84.9% mutation score (vs. 77.9% for Mutahunter), suggesting its mutants explore code regions relevant to real faults, though only a fraction match the exact fault taxonomy. These results provide initial evidence, within the scope of the subjects evaluated, that LLM-based mutation with equivalence triage is feasible for Python.

Keywords

Mutation Testing, Large Language Model, Python, Mutant Generator

1 Introduction

Mutation testing is an established approach for assessing the quality of automated test suites. The technique consists of introducing, in a controlled manner, small changes into the production code, called mutants, and running the test suite to check how many of these changes are detected. Undetected mutants, called surviving mutants, indicate potential weaknesses in the test suite. This approach has been widely studied and refined in the Software Engineering area [11][7][9][8].

Mutant generation has been carried out through classical mutation operators, defined from syntactic patterns identified in the code, such as replacements of arithmetic, conditional, and boolean operators, as well as the insertion or removal of statements and expressions. Although effective, this strategy has relevant limitations. Among them are the computational cost associated with executing large numbers of mutants and the presence of equivalent mutants. These mutants, although syntactically distinct from the original code, do not change its observable behavior and therefore tend to survive the test suite. As a consequence, they can artificially inflate the number of surviving mutants and distort the *mutation score*, hindering the interpretation of the results.

Large language models (LLMs) have begun to be explored as an alternative for mutant generation [5][10][12][4]. Unlike classical operators, LLMs are not restricted to a fixed set of syntactic transformations. This characteristic allows a broader exploration of the production code and the generation of mutants that are potentially more diverse and semantically closer to real faults. Recent studies [5][10][12] indicate that LLM-generated mutants can capture defect patterns that are not directly represented by traditional operators, although they may also introduce noise, such as non-executable, duplicate, or equivalent mutants.

Despite these advances, there is still little evidence on how LLM-based mutation approaches behave across different programming languages. Most studies [5][10][12][4] focus on languages such as Java, JavaScript, and TypeScript, leaving open the generalization of the findings to other ecosystems. This limitation is particularly relevant for Python, whose dynamic typing, runtime interpretation, and dependence on the execution context may affect the validity, executability, and equivalence of the generated mutants. Thus, it remains poorly understood to what extent LLM-based approaches are able to generate relevant mutants for Python programs.

To address this gap, we present LLM-Pyro[1], an approach inspired by LLMorpheus [10] for mutant generation in Python. The goal of this work is to characterize the trade-offs of LLM-based mutation testing in a dynamically typed language and evaluate the performance of the proposed approach in comparison with classical mutation tools for Python, such as mutmut, and with LLM-based tools, such as Mutahunter, an open-source and language-agnostic approach. By focusing on Python, this study investigates whether the benefits and limitations observed in prior work on LLM-based mutation hold in a language with distinct semantic and dynamic characteristics.

The evaluation is conducted using three complementary datasets. HumanEval is used in a pilot study, aiming to compare, on a reduced and controlled scale, the taxonomy of mutants generated by different approaches. The BugsInPy [13] and PyBugHive [1] datasets, composed of real, manually validated, and reproducible faults from open-source Python libraries, are used to assess the ability of the approaches to reproduce real defects. Combining these datasets allows analyzing the generated mutants both in controlled tasks and in scenarios derived from real faults, broadening the diversity of projects, defect types, and execution contexts considered.

[1] https://github.com/VictorManuelChang/LLM-Pyro

This paper contributes: (i) an LLM-based approach for mutant generation in Python; (ii) a comparative evaluation between classical mutation, language-agnostic LLM-based mutation, and the proposed approach; and (iii) an empirical analysis of the relationship between surviving mutants and real faults in Python datasets. With this, the study seeks to advance the understanding of the limits and opportunities of LLM-based mutation in dynamically typed languages.

2 Illustrative Example

To motivate the problem investigated in this work, consider a Python function responsible for computing the final value of a purchase based on the customer type, the total amount, and an optional coupon. Although simplified, this example represents a common pattern in real systems: business rules dependent on conditions, optional values, and normalization of textual inputs.

```
def calculate_total(total, customer_type, coupon=None):
    if total < 0:
        raise ValueError("total must be positive")

    discount = 0.0

    if customer_type == "student":
        discount = 0.10
    elif customer_type == "vip" and total >= 100:
        discount = 0.15

    if coupon and coupon.strip().upper() == "FREESHIP":
        return round(total * (1 - discount), 2), True

    return round(total * (1 - discount), 2), False
```

A classical mutation tool can generate local changes from predefined operators, such as replacing `>=` with `>` in the condition `total >= 100`, swapping `and` for `or`, or modifying numeric constants used in the discount computation. Such changes are useful to assess whether the test suite covers boundaries, logical operators, and expected values. For example, if the suite lacks a test for a `vip` customer with exactly `total = 100`, the mutant that changes `total >= 100` to `total > 100` may survive, indicating a gap in boundary-behavior coverage.

However, some plausible faults in Python are not necessarily captured by simple classical operators. A language model, by considering the function's context, could generate mutants such as removing the coupon normalization, replacing `coupon.strip().upper() == "FREESHIP"` with `coupon == "FREESHIP"`, or changing the handling of optional values, replacing `if coupon and ...` with an expression that assumes `coupon` always contains a string. These mutants exploit semantic aspects of the code, such as input normalization, `None` values, whitespace sensitivity, and string comparison. Such aspects are particularly relevant in Python, due to its dynamic typing and dependence on runtime behavior.

This example illustrates the central trade-off investigated in this work. On the one hand, LLM-based approaches can generate more contextual mutants that are potentially closer to real defects, since they are not limited to a fixed catalog of syntactic operators. On the other hand, this freedom can also introduce noise, producing duplicate, invalid, non-executable, or equivalent mutants. Thus, the question is not only whether LLMs can generate mutants for Python, but to what extent these mutants are valid, diverse, and useful when compared to those generated by classical tools and by language-agnostic LLM-based approaches.

From this scenario, this study empirically evaluates three mutant-generation approaches for Python. The goal is to analyze how they differ regarding the taxonomy of the produced mutants and their ability to generate mutants related to real faults catalogued in Python datasets.

3 LLM-Pyro

Inspired by the approach introduced by LLMorpheus [10], we fully transpiled the open-source repository, originally written in JavaScript, to Python. Our approach, however, presents some differences with respect to the original proposal.

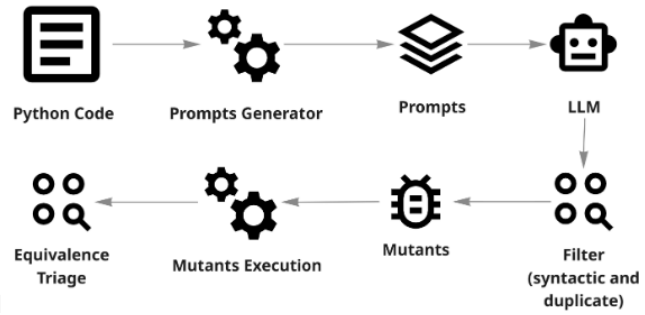

Figure 1: LLM-Pyro workflow

LLM-Pyro (Figure 1) is an approach entirely based on an LLM, without any optimization or training process over the project under test. We used OpenAI's gpt-4o-mini model to generate the mutants from a set of prompts inherited from the original proposal. These prompts consist of parameterized templates that define the role of the LLM as an expert in mutation testing, the activity to be performed, and the production code to be mutated. The Prompt used is a structured prompt, which explicitly presents the activity, guidelines, and objectives, requests three mutation options per snippet, and shows the expected response format.

Before querying the LLM, a logical component traverses the abstract syntax tree (AST) of the production code and identifies the expressions that are candidates for mutation, such as conditions of conditional and loop statements, loop targets and iterables, return values, binary operations, comparisons, and function calls. The extraction is limited to expressions inside functions and ignores bodies composed exclusively of assertions, preventing mutations in the test harnesses themselves from being counted as coverage gaps. Each candidate snippet is replaced by the <PLACEHOLDER> marker, and an individual prompt is rendered for each occurrence. When the file exceeds two hundred lines, only the region around the marker is included, preserving the context without transmitting the entire file.

The set of prompts is then submitted to the LLM, which generates distinct versions for each snippet marked as <PLACEHOLDER>.

The responses go through a layered filtering before being accepted as valid mutants. Syntactically invalid candidates, verified through ast.parse, as well as those identical to the original code, are discarded. Duplicates are detected by an exact-match criterion: a candidate is discarded when another accepted mutant already occupies the same location (file, start/end line, and start/end column) with the same replacement text. This criterion catches only *syntactic* duplicates; two mutants with different replacement text that happen to be semantically equivalent to each other (e.g., `x + 1` and `1 + x` generated for the same location in separate calls) are treated as distinct and both counted. We return to this limitation, together with the related notion of mutant subsumption, in Threats to Validity. The resulting mutants are applied to the code by textual replacement at the recorded positions and executed by the test suite.

During execution, each mutant is classified as killed, when some test fails by assertion or by any runtime exception, or as survived, when it passes the entire suite without detection. The surviving mutants are collected and the mutation score metric is computed internally by a Python component. At this stage there is no distinction between equivalent and non-equivalent mutants, so all valid mutants are considered in the computation, according to the formula:

$$\textit{mutation score} = \begin{cases} 0 & \text{if } |R| = 0 \\ \dfrac{|\{\, r \in R \mid r.\textit{status} = \textsc{killed} \,\}|}{|R|} & \text{otherwise} \end{cases} \tag{1}$$

where $R = \{\, r \in \textit{results} \mid r.\textit{status} \in \{\textsc{killed}, \textsc{survived}\} \,\}$.

After this stage, the set of surviving mutants is forwarded to an LLM-based Oracle. For each mutant, the Oracle receives the original code, the mutated snippet, and the lines around the mutation point, which allows it to reason about the surrounding structure. Following a guided chain of reasoning, the model identifies what was changed, checks known equivalence patterns, and tries to construct an input capable of distinguishing the mutant's behavior from the original, ultimately classifying it as equivalent, non-equivalent, or unknown. This stage distinguishes LLM-Pyro from the original proposal by integrating, within a single pipeline, LLM-based mutant generation and equivalence triage, separating useful mutants from the noise introduced by equivalent ones.

Algorithm 1 summarizes the complete flow of the approach, from the extraction of candidates in the AST to the equivalence triage of the survivors.

Algorithm 1 LLM-Pyro pipeline

Require: Source code S, test suite T, LLM model M
Ensure: Mutants with *status* (killed/survived) and equivalence verdict

$C \leftarrow \textsc{ExtractCandidates}(S)$
$P \leftarrow \textsc{GeneratePrompts}(C)$
$\textit{Mutants} \leftarrow \emptyset$
for each *prompt* $p \in P$ **do**
  $R \leftarrow M.\textsc{query}(p)$
  for each replacement $s \in R$ **do**
    if $s = p.\textit{original}$ **or** $\neg\textsc{ValidSyntax}(s)$ **or** $s \in \textit{Mutants}$ **then**
      discard
    else
      $\textit{Mutants} \leftarrow \textit{Mutants} \cup \{\textsc{ApplyMutation}(S, p, s)\}$
    end if
  end for
end for
$\textit{Survivors} \leftarrow \emptyset$
for each $m \in \textit{Mutants}$ **do**
  if $\textsc{RunTests}(T, m)$ all pass **then**
    $m.\textit{status} \leftarrow \textsc{survived}$; $\textit{Survivors} \leftarrow \textit{Survivors} \cup \{m\}$
  else
    $m.\textit{status} \leftarrow \textsc{killed}$
  end if
end for
for each $m \in \textit{Survivors}$ **do**
  $m.\textit{verdict} \leftarrow \textsc{Oracle}(M, m)$
end for
return *Mutants*

4 Evaluation

The evaluation consists of a comparative empirical study, structured in two phases, with the goal of investigating whether LLM-based mutation generalizes to Python considering its dynamic typing. To this end, LLM-Pyro's performance is compared against two reference tools: mutmut, based on classical operators, and Mutahunter, based on LLMs.

4.1 Research Questions

Accordingly, this work evaluates the proposed approach through the following research questions:

RQ1 What is the taxonomy of surviving mutants considering the language's characteristics?
RQ2 To what extent do LLM-Pyro's mutants explore the locations and taxonomies of real faults in Python?
RQ3 How many surviving mutants are equivalent?
RQ4 What is the overall cost and execution time of the approach?

4.2 Experimental Setup

4.2.1 Research Design. The experiment is designed around four complementary evaluation dimensions, each aligned with one of the research questions introduced above: generation quality and taxonomy of the produced mutants (RQ1), proximity location and taxonomy to real faults (RQ2), equivalence triage among survivors (RQ3), and cost (RQ4). Three treatments are compared across these dimensions: LLM-Pyro, our adaptation of LLMorpheus [10] for Python; Mutahunter [3], an open-source tool that implements a language-agnostic LLM-based mutation approach; and mutmut, which represents mutation based on classical operators.

Phase 1 — Controlled pilot study. We use HumanEval [2], a controlled *benchmark* composed of classic, low-complexity problems, in a format similar to that of platforms such as LeetCode and Beecrowd. Each problem provides a reference solution, which constitutes the code to be mutated, and a test suite, which is executed to evaluate

the mutants. The goal of this phase is to feasibly collect metrics such as the taxonomy of the mutants, the *mutation score*, and equivalence triage. Because they do not require complex dependencies, these problems allow running the vast majority of cases with the three treatments, enabling a direct comparison between them.

Phase 2 — Evaluation on real faults. We use the BugsInPy [13] and PyBugHive [1] datasets, with the purpose of assessing to what extent the generated mutants resemble real faults. In this phase, mutation is restricted to the exact lines of the fix *patch* of each bug, so that the mutants produce variants located precisely in the snippet that was changed to fix the defect. Then, the mutants generated over the fixed version of the code are compared with the version in which the defect was originally reported, contrasting both the taxonomy and the location of the mutation, in order to assess whether the mutants are similar or identical to the catalogued fault.

4.2.2 Dataset Selection. The HumanEval dataset was used in the pilot experiment stage, with the purpose of comparing the taxonomy of mutant generation across approaches in a controlled and smaller-scale scenario. The use of BugsInPy and PyBugHive, in turn, aims to assess the effectiveness of the approaches in reproducing real faults, that is, in generating mutants whose nature is close to the faults actually observed in production projects.

BugsInPy [13], inspired by Defects4J [6], gathers 493 real faults extracted from 17 open-source Python projects, selected so as to represent different application domains, such as machine learning, development tools, scientific computing, and web frameworks. All faults are reproducible and isolated, and were manually validated by at least two independent reviewers. PyBugHive PyBugHive [1], in turn, provides 149 real faults, likewise manually and reproducibly validated, from 11 popular Python projects on GitHub. Table 1 summarizes the number of faults and repositories covered by each dataset.

Table 1: Number of faults and repositories in each real-fault Python dataset used in this work.

Dataset	# Faults	# Repositories
BugsInPy	493	17
PyBugHive	149	11
Total	**642**	**28**

4.2.3 Language Model Selection. The textitgpt-4o-mini was selected primarily for its balance between efficient mutant generation and low cost per API call, being a smaller, faster, and cheaper variant of gpt-4o. Beyond cost, it is the same proprietary model used in the original model [10] approach for JavaScript and TypeScript, so adopting it here enables cross-language comparison of results and supports analysis of how well the LLM-based mutant generation technique generalizes beyond its originally evaluated ecosystem.

4.2.4 Reproducibility Details. All experiments were executed between May and July 2026, using Python 3.10 (the project requires Python ≥3.9) and, for the two LLM-based approaches, the *gpt-4o-mini* model accessed through an OpenAI-compatible chat-completions endpoint. Table 2 lists the API parameters held fixed across all LLM-Pyro runs, beyond the temperature discussed above. In Phase 2 (real-fault datasets), the placeholder window is additionally constrained to the exact lines touched by the fix patch (`max_lines_in_placeholder`), whereas Phase 1 (HumanEval) applies no such restriction. All raw configuration values are recorded per run in the `metaInfo` block of each experiment's `summary.json`, part of the public artifact.

Table 2: API and generation parameters held fixed across LLM-Pyro runs.

Parameter	Value
Model	gpt-4o-mini
Temperature	0
Max output tokens per call	250
Retry attempts on failure	3
Context window around placeholder	±100 lines (max 200)
Mutants requested per snippet (full template)	3

4.3 RQ1: What Is the Taxonomy of Surviving Mutants Considering the Language's Characteristics?

To characterize *what* survives the test suite, we classify each surviving mutant from the pilot experiment (HumanEval) according to a taxonomy of eight categories, derived from the mutation point in the AST: boundary change (*boundary*), operator swap (*operator*), variable substitution (*variable*), constant/literal change (*constant*), function-call modification (*function*), condition change (*condition*), return value change (*return_val*), and other cases (*other*). We adapted these categories from the mutation operators of `mutmut` and Petrovick et all [8], reorganizing them around the AST node at which each mutation is applied to make classification more suitable for our analysis. Classification was applied, with the same component, to the survivors of mutmut and Mutahunter, allowing us to compare not only how many mutants each approach lets through, but also the *nature* of those survivors. Table 3 presents the distribution. LLM-Pyro leaves 154 surviving mutants, against

Table 3: Taxonomy of surviving mutants per approach (HumanEval, 40 problems).

Category	LLM-Pyro	mutmut	Mutahunter
Constant (*constant*)	53	21	5
Operator (*operator*)	34	3	0
Function (*function*)	17	1	1
Boundary (*boundary*)	13	11	10
Variable (*variable*)	6	1	4
Condition (*condition*)	5	0	2
Return (*return_val*)	5	1	7
Other (*other*)	5	0	2
Equivalent	16	1	4
Total survivors	**154**	**39**	**35**

39 for mutmut and 35 for Mutahunter, a direct consequence of generating a substantially larger volume of mutants. More relevant than volume, however, is the *diversity* of the categories. LLM-Pyro's survivors are distributed across all eight categories, with a notable concentration in constant changes (53), operator swaps (34), and function-call modifications (17). mutmut, in contrast, concentrates its survivors in constants (21) and boundaries (11), reflecting its fixed set of syntactic operators: it produces almost no survivors through operator swaps (3), function changes (1), or condition changes (0). Mutahunter, although also LLM-based, generates few survivors overall and none in the operator category.

The predominance of survivors through operator swaps and function-call modifications in LLM-Pyro is the most significant finding for the Python language: these are precisely the mutations that exploit dynamic semantics that mutmut's fixed syntactic operators cannot reach. These survivors correspond to coverage gaps that only manifest when the mutant changes the program's *behavior* in a way the suite does not distinguish, not merely its syntax. This supports the hypothesis that LLM-based generation expands the space of relevant mutations beyond what classical tools can cover.

4.3.1 AST Distance. Beyond the different taxonomies, the distribution of mutants also differs across approaches. To characterize the structural magnitude of the mutations, we employ the *Zhang-Shasha Tree Edit Distance* (TED) [14], with unit cost, between the AST subtree of the original fragment and that of the mutated fragment. The analysis was conducted in two stages: a pilot study on HumanEval, composed of simple and controlled activities, and an evaluation on PyBugHive, composed of real Python libraries with actually observed faults.

Pilot study (HumanEval).

Before analyzing which mutants survive the test suite, we characterize the generation quality in the pilot study (HumanEval, 40 problems), which underpins the compilability, duplication, and equivalence rates cited in the abstract. Table 4 details the generation funnel: of the 1,148 unique candidates, 829 (72.2%) are compialable, with a 15.3% duplication rate among them and only 16 equivalent mutants (10.4% of survivors). These rates are consistent with prior reports for Java/JavaScript, suggesting that such quality costs may be characteristic of LLM-based generation rather than solely attributable to Python's dynamic typing.

Table 4: Mutant generation funnel in the pilot (HumanEval, 40 problems).

Metric	Count	Rate
Unique candidates generated	1,148	—
Compilable (valid)	829	72.2%
Non-compilable (discarded)	319	27.8%
Identical to original	0	0.0%
Duplicates (discarded)	127	15.3%
Survivors	154	—
Equivalent (Oracle)	16	10.4%

Table 5: Number of mutants (n) and mean AST edit distance per repository, for LLM-Pyro and Mutahunter, on the PyBugHive set.

	LLM-Pyro		Mutahunter	
Repository	n	Mean	n	Mean
black	674	3.13	45	3.00
cookiecutter	155	2.90	5	2.80
discord.py	119	3.01	3	2.33
poetry	218	2.94	23	3.17
scrapy	147	2.62	6	2.50

After this stage, we analyzed the compilable LLM-Pyro mutants. The TED distribution is wide (minimum 0, maximum 22, mean 2.96, median 3): only about 25% of mutations are structurally trivial (TED equal to 0, such as swapping a literal or renaming an identifier), while approximately 60% have a TED greater than or equal to 2, with a long tail up to 22. This profile contrasts with that of classical operators, which concentrate at TED equal to 1, and shows that the LLM performs structural rewrites rather than only pointwise substitutions. The mutation is also asymmetric: the model inserts more structure than it removes, introducing constructs absent from the original, such as slicing and arithmetic operations. This is reflected in the most frequent root-node transitions, where simple variables (`Name`) are replaced by richer expressions, such as binary operations (`BinOp`), subscripts (`Subscript`), and calls (`Call`). Among surviving mutants, the mean TED is slightly higher (3.21) than that of killed mutants (2.91), suggesting that mutations that escape the suite tend to be structurally deeper.

Evaluation on real faults (PyBugHive). In the next stage, TED analysis was applied to the mutants generated by the two LLM-based approaches on real libraries, directly comparing LLM-Pyro and Mutahunter. As summarized in Table 5, the two tools generate mutations of similar syntactic magnitude, with a mean TED close to 3 per repository (between 2.6 and 3.2). The marked difference lies in sampling density: LLM-Pyro produces a considerably larger number of mutants per repository, whereas Mutahunter generates few mutations in most projects (for example, 3 in `discord.py` and 5 in `cookiecutter`, against 119 and 155 for LLM-Pyro, respectively; and 45 against 674 in `black`). Thus, while structural magnitude per mutation is maintained, LLM-Pyro offers denser sampling of the code's syntactic neighborhood, expanding the coverage of mutable points per file.

RQ1. Insight. These results suggest that the main contribution of LLM-Pyro is not merely producing more mutants, but exposing a broader spectrum of test-suite weaknesses. While classical mutation remains concentrated on constants and boundary changes, LLM-Pyro produces surviving mutants across all taxonomy categories, especially operators and function calls. This indicates that LLM-based mutation can exercise Python-specific semantic aspects, such as overloaded operators, polymorphic calls, implicit conversions, and runtime-dependent behavior, which are less likely to be reached by fixed syntactic operators.

4.4 RQ2: To What Extent Do LLM-Pyro's Mutants Explore the Locations and Taxonomies of Real Faults in Python?

To assess whether the mutants generated by LLM-Pyro explore the same regions and categories as real faults, we use PyBugHive, which contains 149 manually validated bugs from 11 open-source Python projects. For each bug, we restrict mutation to the exact lines of the fix patch, so that mutants produce variants located precisely in the region that was changed to fix the defect. This restriction operationalizes proximity to real faults in terms of location: it simulates a scenario in which the test suite must detect semantically relevant changes in the region of code that contained the fault, without implying full semantic equivalence to the original bug.

We run LLM-Pyro and Mutahunter on 44 bugs from 4 projects (black, cookiecutter, discord.py, and scrapy).[2] Table 6 summarizes the results.

Table 6: Comparison between LLM-Pyro and Mutahunter on real faults (PyBugHive, 44 bugs, 4 projects).

Tool	Mutants	Killed	Survived	Score
LLM-Pyro	1,101	935	166	84.9%
Mutahunter	86	67	19	77.9%
LLM-Pyro generates 12.8× more mutants; score is 7 p.p. higher.				

LLM-Pyro generated 1,101 effective mutants (*killed* + *survived*), against only 86 for Mutahunter, a 12.8:1 ratio. LLM-Pyro's mutation score was 84.9%, against 77.9% for Mutahunter. This volume difference stems from the generation strategy: LLM-Pyro sends one prompt per target line and aggregates the results of multiple calls, whereas Mutahunter sends the patch excerpt in a single call with a limited output-token budget. When the patch's line range spans hundreds of lines, Mutahunter fails to generate focused mutants: 14 of 51 tasks produced zero mutants, against only 4 for LLM-Pyro.

This volume advantage is statistically significant: LLM-Pyro generates, on average, about 21.8× more total mutants than Mutahunter (paired Wilcoxon test, rank-biserial correlation = 1.000; $p = 0.016$) and about 14.3× more surviving mutants (rank-biserial correlation = 0.929; $p = 0.016$). However, when we examine the dispersion of surviving mutants, LLM-Pyro's standard deviation exceeds its own mean, a pattern also observed in the other datasets, indicating that although it produces more mutants per activity, its survivors are not evenly distributed across activities. Mutahunter, in contrast, shows values close to one another in most cases, reflecting more uniform generation, albeit considerably more limited in volume.

It is worth noting that this advantage reflects the generation strategy implemented by LLM-Pyro (one prompt per target line) rather than necessarily the intrinsic capability of the underlying model, which is the same (gpt-4o-mini) for both tools. The comparison therefore evaluates the two tool configurations as used, and not LLM-Pyro and Mutahunter as abstract concepts.

[2] poetry was excluded from this comparison because its test dependencies (`pytest`) were not available in the configured environment, resulting in exit code 127 for all mutants, which would make any mutation score for this project invalid.

When we analyze the taxonomy of the mutants generated by both approaches and relate it to the real faults reported in the dataset, we find that the two approaches perform similarly in terms of taxonomy-matching rate: in only 18.2% of cases does at least one surviving mutant fall into the same taxonomy category as the reported fault. This taxonomy match is a coarse indicator of alignment; a mutant sharing a fault's category (e.g., both being condition changes) does not imply that it reproduces the fault's behavior. Even so, LLM-Pyro explores a broader spectrum of these taxonomies than Mutahunter, as shown in Table 7. Generating mutants distributed across a larger number of categories is therefore LLM-Pyro's main advantage in this respect.

Of LLM-Pyro's 166 survivors, no equivalence oracle was run on this dataset (a task left for future work). These survivors represent residual coverage gaps in the test suites of the evaluated projects.

As a second real-fault dataset, we also ran BugsInPy on bugs from the *youtube-dl* project. Restricting the analysis to the files actually exercised by the offline test suite (`utils.py`, `YoutubeDL.py`, `jsinterp.py`, and `common.py`; the extractors depend on network access and are not covered by tests, killing zero mutants under any approach), LLM-Pyro generated 1,411 effective mutants, against 1,033 for mutmut and 98 for Mutahunter. LLM-Pyro's mutation score on these files was 69.1%, lower than mutmut's 83.8%. This result cautions against interpreting raw mutation score in isolation: LLM-Pyro generates ≈1.4× more mutants than mutmut, and, consistent with the diversity observed in RQ1, a larger fraction of them survive. One plausible interpretation is that these survivors are harder-to-kill mutants that exploit the language's dynamic semantics and expose gaps the existing suite does not distinguish; however, we did not conduct a qualitative analysis confirming this coupling, so the lower raw score should not be read, on its own, as evidence of either weaker or stronger performance.

RQ2. Insight. The results on real faults indicate that LLM-Pyro provides a denser exploration of code regions associated with actual bug fixes rather than direct reproduction of those faults. Its higher mutant volume and competitive mutation score suggest that LLM-based mutation can generate many behaviorally relevant variants around real-fault locations, and it covers a broader range of fault taxonomies than Mutahunter, even though only 18.2% of cases match the exact taxonomy of the reported fault. The BugsInPy/youtube-dl result further cautions against interpreting raw mutation score in isolation: one plausible interpretation is that the lower score reflects harder-to-kill mutants revealing residual coverage gaps, but this remains a hypothesis that would require a qualitative analysis of surviving mutants to confirm.

4.5 RQ3: How Many Surviving Mutants Are Equivalent?

Before reporting rates, we note that the Oracle performs *triage*, not ground-truth equivalence classification: its verdicts are LLM-based judgments intended to separate likely equivalents from likely real gaps, not a verified ground truth. All percentages below should be read under this framing.

Of the 154 mutants that survive the test suite in HumanEval, 16 are triaged by the Oracle as equivalent, corresponding to an equivalence rate of 10.4%. These mutants represent false positives

Table 7: Comparison between LLM-Pyro and Mutahunter regarding the exploration of the taxonomy of catalogued real faults (PyBugHive), restricted to the 11 activities common to both tools.

		LLM-Pyro			Mutahunter			
Activity	Fault taxonomy	Surv.	Cat. count	Explores?	Surv.	Cat. count	Explores?	Advantage
cookiecutter_1513_config	Variable	2	0	No	0	0	No	None
cookiecutter_18_generate	Function	42	10	Yes	2	2	Yes	Both
cookiecutter_18_setup	Constant	0	0	No	4	1	Yes	Mutahunter
discord.py_7676___init__	Operator	5	0	No	1	0	No	None
discord.py_7818_commands	Condition	14	6	Yes	1	0	No	LLM-Pyro
poetry_3224_locker	Function	0	0	No	0	0	No	None
poetry_3263_locker	Condition	0	0	No	0	0	No	None
poetry_3263_provider	Variable	0	0	No	0	0	No	None
scrapy_1265_conf	Return	16	0	No	1	0	No	None
scrapy_1265_deprecate	Other	0	0	No	1	0	No	None
scrapy_2552___init__	Condition	7	0	No	0	0	No	None

produced by the approach: syntactic modifications that do not change the program's observable behavior for any possible input and, therefore, do not indicate a real gap in the test suite.

The 90 mutants triaged as non-equivalent (58.4%) represent real coverage gaps, cases in which the mutated program would produce a different output for some input, but the existing test suite failed to detect that difference. The adjusted mutation score, computed by excluding Oracle-triaged equivalents from the denominator, rises from 81.4% to 83.0%, reducing by 1.6 percentage points the penalty imposed by these false alarms.

The remaining 48 survivors (31.2%) could not be triaged with sufficient confidence and are labeled *unknown* by the Oracle. Rather than a shortcoming of the triage pipeline, we treat this outcome as a deliberate methodological choice: when the model cannot construct a distinguishing input or is uncertain about a mutant's behavioral effect, it withholds a verdict instead of forcing a binary decision. This conservative design avoids false precision, an oracle that always returns *equivalent* or *non-equivalent* would inflate confidence in results that are, in fact, uncertain, which is a more serious threat to validity than an explicit *unknown* category. The *unknown* label thus establishes a more honest boundary between what the Oracle can triage automatically and what still requires human review, supporting its usefulness even without additional human validation in this study, provided the reported rates are read as conservative bounds rather than as overconfident point estimates.

The Oracle-triaged equivalence rate of 10.4% is in line with the values reported for LLM-based tools on the same dataset (11.4% for Mutahunter) and substantially higher than the 2.6% for mutmut. This result supports the hypothesis that a higher equivalence rate is an inherent characteristic of LLM-based generation, which rewrites code freely and occasionally produces semantically identical versions (e.g., `abs(a-b)` → `abs(b-a)`), rather than an artifact specific to LLM-Pyro or to Python.

RQ3. Insight. The equivalence results show that surviving mutants should not be treated as a homogeneous set of test-suite weaknesses. A meaningful portion of survivors is triaged as equivalent, while another portion remains uncertain. Equivalence triage is therefore not a secondary implementation detail, but a necessary step for making LLM-based mutation testing interpretable and actionable. The *unknown* category is also an important finding in its own right: rather than a limitation to be minimized, it reflects a conservative design choice that avoids false precision, and it creates a more honest boundary between automated triage and human review, one that helps justify the Oracle's value even in the absence of additional human validation.

4.6 RQ4: What Is the Overall Cost and Execution Time of the Approach?

The approach uses the *gpt-4o-mini* model both for mutant generation and for the equivalence Oracle. Cost is measured in tokens consumed by API calls, according to prevailing prices (US$ 0.15 per 1 M input tokens and US$ 0.60 per 1 M output tokens). Table 8 summarizes the consumption and cost data per experiment. The total cost of all runs performed is approximately US$ 0.66, demonstrating the economic feasibility of the approach even with proprietary models. The Oracle was run only on the HumanEval survivors (154 calls); for the other datasets, this step was not executed in this preliminary evaluation.

Table 8: Tokens consumed and estimated cost per experiment. The equivalence Oracle was run only on the 154 HumanEval survivors.

Experiment	Input tokens	Output tokens	Cost (US$)
HumanEval — generation	124,614	42,299	0.04
HumanEval — Oracle	68,966	53,323	0.04
BugsInPy — generation	1,692,331	181,291	0.36
PyBugHive — generation	1,021,469	116,262	0.22
Total	**2,907,380**	**393,175**	**0.66**

Regarding execution time, PyBugHive was run in a single session and recorded a wall-clock time of approximately 73 minutes for 55 bugs across 5 distinct projects (black, cookiecutter, scrapy, discord.py, poetry), corresponding to ≈ 1.3 min/bug on average. In BugsInPy, runs were distributed across multiple sessions over 9 calendar days; analyzing individually the bugs executed in a continuous session, the typical time per bug ranged from 1 to 5 minutes, depending on the size of the mutated file and the test suite. In HumanEval, execution time per problem was likewise on the order of minutes, dominated by the cost of API invocation and the sequential execution of each mutant.

RQ4. Insight. The cost results suggest that the financial barrier for LLM-based mutation testing may be lower than expected, especially when using a smaller model such as gpt-4o-mini. The main practical bottlenecks appear to be execution time, test-environment configuration, and the interpretation of surviving mutants rather than API cost itself. This makes LLM-Pyro potentially viable for targeted use in critical modules, provided that teams adopt filtering, equivalence triage, and selective review workflows.

5 Threats to Validity

Internal validity. The non-deterministic nature of LLMs may introduce variation across runs; we mitigate this effect by adopting temperature 0, at which repeated runs tend to produce the same set of mutants [10]. The environment configuration of the real faults is also a threat: in PyBugHive, the *poetry* project had to be excluded from the comparison because its test dependencies were not installed correctly (*exit code* 127 for all mutants), which would invalidate any *mutation score*; analogously, in BugsInPy, the *youtube-dl extractors* depend on network access and are not exercised by the offline suite, so we restricted the analysis to the files that are effectively tested. These cases reinforce that the metric is only meaningful when the suite actually executes the mutated code.

Construct validity. The *mutation score* is a *proxy* for test-suite quality and does not directly capture coupling to real faults. In particular, in the BugsInPy/*youtube-dl* case (RQ2), LLM-Pyro's lower raw score relative to mutmut is only tentatively attributed to harder-to-kill mutants; without a qualitative analysis contrasting survivors with real fixes, this remains a hypothesis rather than a confirmed explanation, and raw mutation score should not be compared across tools without accounting for mutant composition. Moreover, the equivalence Oracle itself is LLM-based, and we did not perform additional human validation of its individual verdicts. However, 31.2% of the HumanEval survivors were classified as *unknown* rather than forced into an equivalent/non-equivalent decision; we treat this as a conservative design choice that limits false precision rather than as evidence of unreliability, though it does mean the reported equivalence rate of 10.4% should be read as a lower bound, and the non-equivalent rate of 58.4% as an upper bound, pending future validation.

External validity. The evaluation is preliminary and limited in scope. The pilot study uses HumanEval, whose problems are isolated, synthetic, and possibly present in the model's training data (contamination), which may inflate the generation capability. The evaluation on real faults covers a subset of projects (4 from PyBugHive and *youtube-dl* from BugsInPy), insufficient to generalize to the entire Python ecosystem. Finally, all experiments use a single model (*gpt-4o-mini*); different models may produce different volumes, compilability rates, and equivalence rates.

Conclusion validity. The comparison against Mutahunter may appear unfair, since a large part of the observed difference stems from the generation strategy as implemented, and not from the intrinsic conceptual capability of each approach. LLM-Pyro issues one prompt per target line and aggregates the results of multiple calls, whereas Mutahunter sends the patch snippet in a single call with a limited output-token budget, which penalizes it precisely when the patch spans many lines. Both tools use the same model (*gpt-4o-mini*); therefore, our results should be read as a comparison of the two tool configurations *as used*, and not as a verdict on LLM-Pyro and Mutahunter as abstract concepts. A comparison under equalized prompting and token budgets is left for future work.

6 Related Work

Mutation testing and the cost of equivalent mutants. Mutation testing is a well-established technique whose benefits and challenges have been validated in industrial contexts [7–9]. Prior work shows that it leads developers to write more and higher-quality tests [7], proposes AST-based arid nodes to reduce unproductive mutants and generation time [8], and examines how resolving surviving mutants during code review affects quality [9]. Despite these positive impacts, the main obstacle to large-scale adoption remains the triage of equivalent mutants: syntactic modifications that do not change observable behavior and thus do not represent real gaps in the test suite. Classical Python tools such as mutmut apply a static operator set with no native treatment for this issue, delegating manual analysis to the developer.

Mutant generation with LLMs. Several studies employ language models to generate more expressive mutants [5, 10, 12]. LLMorpheus [10], which directly inspires our approach, uses *prompt templates* to mutate JavaScript and TypeScript, while BugFarm [5] synthesizes complex defects to evaluate fault-prediction models. A comprehensive study on Java [12] reveals a clear trade-off: LLMs generate mutants behaviorally closer to real faults, raising the detection rate, but at the cost of higher non-compilability, duplication, and equivalence rates. On the tooling side, Mutahunter [3] offers an open-source, language-agnostic alternative. LLM-Pyro advances beyond these works by investigating whether this same trade-off manifests in Python under dynamic typing and by integrating an automated equivalence-triage stage absent from LLMorpheus and Mutahunter.

Real-fault datasets. Assessing the coupling of mutants to authentic defects relies on controlled repositories. Inspired by Defects4J [6] in the Java ecosystem, datasets such as BugsInPy [13] and PyBugHive [1] provide validated, reproducible *bugs* in open-source Python projects, making it possible to verify empirically whether generated mutants reflect the nature and location of real fixes.

Positioning. LLM-Pyro fills a gap in the literature by adapting LLM-based mutation to Python and coupling an oracle to mitigate the equivalence problem, standing out by contrasting its results simultaneously with existing tools (mutmut and Mutahunter) and with real-fault repositories.

7 Conclusion and Future Work

We presented LLM-Pyro, a Python adaptation of LLMorpheus that couples mutant generation with an LLM-based equivalence oracle, offering an initial empirical characterization, within the subjects evaluated, of the trade-offs of LLM-based mutation testing in a dynamically typed language: mutant diversity, equivalence rate, generation cost, and behavior on real-world code. Our preliminary evaluation indicates that LLM-based mutation transfers to Python with quality costs consistent with those reported for Java/JavaScript

(72.2% compilability, 15.3% duplication, and 10.4% equivalence). As we do not perform a controlled cross-language comparison, we read this as suggesting that such costs may be characteristic of LLM-based generation rather than solely attributable to Python's dynamic typing.

Where dynamic typing does appear is not in these generation-quality costs but in *what kind of survivor* LLM-Pyro produces: its survivors concentrate in the operator and function-call categories that mutmut's fixed operator set barely reaches, as illustrated by runtime-resolved, type-independent rewrites such as `is not None` → `!= None` and an encoding literal changed from `'ascii'` to `'utf-8'`, which a fixed catalog of syntactic operators is not designed to produce. Confirming this as a general property of Python, rather than of these specific subjects, would nonetheless require broader evaluation.

On HumanEval, LLM-Pyro generated more mutants than mutmut and Mutahunter, with survivors spread across a wider range of categories. On PyBugHive, it achieved an 84.9% mutation score while generating 12.8x more mutants than Mutahunter, though alignment with the exact taxonomy of the reported fault remained partial (18.2% of cases). The Oracle produced a non-*unknown* verdict for 68.8% of HumanEval survivors, raising the adjusted score from 81.4% to 83.0%, while conservatively withholding a verdict on the remaining 31.2% rather than forcing an unreliable one. The entire set of experiments cost approximately US$0.66, suggesting the approach may be economically viable at this experimental scale.

Taken together, these findings suggest that, within the subjects evaluated, LLM-based mutation testing for Python should be understood not merely as a way to increase mutant volume or mutation score, but as a means of exposing semantically richer gaps in test suites, provided that generation is coupled with filtering, equivalence triage, and careful interpretation of surviving mutants. Given the limited number of projects covered, broader claims about generalization to the Python ecosystem require further evaluation. As future work, we plan to (i) run the equivalence Oracle on surviving mutants from real-fault datasets such as PyBugHive and BugsInPy, currently limited to HumanEval; (ii) extend the evaluation to more projects while addressing environment limitations involving native or network-dependent dependencies; (iii) compare different language models; and (iv) conduct a qualitative analysis that directly contrasts surviving mutants with real-fix diffs to measure fault coupling more directly than *mutation score* alone.

Artifact Availability

The LLM-Pyro source code, the *benchmark scripts* for HumanEval, BugsInPy, and PyBugHive, and the raw data from the runs (generated mutants, per-task results, and summaries) are publicly available at: https://github.com/VictorManuelChang/LLM-Pyro.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brazil, Finance Code 001, by the Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (FUNDECT, Call No. 42/2024), and by the Federal University of Mato Grosso do Sul (UFMS)

During the preparation of this work, the authors used the Claude language model (Anthropic) as a supporting tool for organizing and formatting the results tables and for style revision. All content was reviewed, edited, and validated by the authors, who take full responsibility for the material presented.

References

[1] Gábor Antal, Norbert Vándor, István Kolláth, Balázs Mosolygó, Péter Hegedűs, and Rudolf Ferenc. 2024. PyBugHive: A Comprehensive Database of Manually Validated, Reproducible Python Bugs. *IEEE Access* 12 (2024), 123739–123756. doi:10.1109/ACCESS.2024.3449106

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[3] CodeIntegrity AI. 2025. Mutahunter: Open-Source Language Agnostic LLM-based Mutation Testing. https://github.com/codeintegrity-ai/mutahunter.

[4] Asma Hamidi, Ahmed Khanfir, and Mike Papadakis. 2025. Intent-Based Mutation Testing: From Naturally Written Programming Intents to Mutants. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 347–357. doi:10.1109/ICSTW64639.2025.10962508

[5] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2025. Challenging Bug Prediction and Repair Models with Synthetic Bugs. In *2025 IEEE International Conference on Source Code Analysis Manipulation (SCAM)*. 133–144. doi:10.1109/SCAM67354.2025.00021

[6] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[7] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does Mutation Testing Improve Testing Practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 910–921. doi:10.1109/ICSE43902.2021.00087

[8] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2022. Practical Mutation Testing at Scale: A view from Google. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3900–3912. doi:10.1109/TSE.2021.3107634

[9] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2023. Please fix this mutant: How do developers resolve mutants surfaced during code review?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 150–161. doi:10.1109/ICSE-SEIP58684.2023.00019

[10] Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing Using Large Language Models. *IEEE Transactions on Software Engineering* 51, 6 (2025), 1645–1665. doi:10.1109/TSE.2025.3562025

[11] Marco Tulio Valente. 2020. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.* Editora: Independente.

[12] Bo Wang, Mingda Chen, Ming Deng, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M. Zhang. 2026. A Comprehensive Study on Large Language Models for Mutation Testing. *ACM Trans. Softw. Eng. Methodol.* (March 2026). doi:10.1145/3805038 Just Accepted.

[13] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. doi:10.1145/3368089.3417943

[14] Kaizhong Zhang and D. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18 (1989), 1245–1262. doi:10.1137/0218082

A Statistical Analysis of Mutant Volume

This appendix reports the full statistical comparison between LLM-Pyro and Mutahunter regarding the number of mutants generated per task, across the three datasets (HumanEval, BugsInPy, and PyBugHive). For each dataset, two measures are considered: the total number of mutants and the number of surviving mutants. Table 9 presents the descriptive statistics, and Table 10 the paired Wilcoxon signed-rank tests, with the one-sided alternative hypothesis that LLM-Pyro produces more mutants than Mutahunter. The effect size is reported as the rank-biserial correlation.

Table 9: Descriptive statistics of the number of mutants generated per task by LLM-Pyro and Mutahunter, per dataset and measure. N is the number of paired tasks; CoV denotes the coefficient of variation.

Dataset	Measure	Tool	Mean	SD	SE	CoV
HumanEval ($N = 38$)	Total mutants	LLM-Pyro	20.92	25.66	4.162	1.226
		Mutahunter	4.605	0.5945	0.09645	0.1291
	Surviving mutants	LLM-Pyro	4.026	7.489	1.215	1.860
		Mutahunter	0.9211	0.9693	0.1572	1.052
BugsInPy ($N = 36$)	Total mutants	LLM-Pyro	44.17	33.74	5.624	0.7640
		Mutahunter	2.750	1.180	0.1967	0.4292
	Surviving mutants	LLM-Pyro	20.83	29.76	4.960	1.428
		Mutahunter	1.417	1.131	0.1885	0.7982
PyBugHive ($N = 7$)	Total mutants	LLM-Pyro	59.29	39.60	14.97	0.6680
		Mutahunter	2.714	0.9512	0.3595	0.3504
	Surviving mutants	LLM-Pyro	12.29	14.36	5.428	1.169
		Mutahunter	0.8571	0.6901	0.2608	0.8051

Table 10: Paired Wilcoxon signed-rank tests comparing the number of mutants generated by LLM-Pyro and Mutahunter (one-sided alternative: LLM-Pyro > Mutahunter). The effect size is the rank-biserial correlation, reported with its standard error (SE) and 95% confidence interval (CI).

Dataset	Measure	W	z	p	Rank-biserial	SE	95% CI
HumanEval	Total mutants	679.5	4.948	$< .001$	0.9331	0.1864	[0.8649, 0.9675]
	Surviving mutants	348.5	3.313	$< .001$	0.7167	0.2130	[0.4968, ∞)
BugsInPy	Total mutants	666.0	5.232	$< .001$	1.000	0.1889	[1.000, 1.000]
	Surviving mutants	528.0	4.937	$< .001$	1.000	0.1998	[1.000, ∞)
PyBugHive	Total mutants	28.00	2.366	.016	1.000	0.3991	[1.000, 1.000]
	Surviving mutants	27.00	2.197	.016	0.9286	0.3991	[0.7411, ∞)