The Testing Showdown: Humans vs LLMs in Bug Detection

Final Project Report

December 3, 2025

Team Number	29
Team Members	Qasim Naik (2025201064)
	Raj K Jain (2025201036)
	S. Praneeth Reddy(2025204006)
	T. Sathvik Reddy (2025201020)
	D. Subhash Chandra (2025201007)
Code Repository	Team 29 Github Link
Presentation	PPT Link
Interactive Dashboard:	Human-vs-LLM

Abstract

This report presents a comprehensive comparative study between two testing strategies applied to a collection of buggy Python functions from the MBPP dataset: (1) *LLM-generated example-based tests* using `pytest`, and (2) *human-written property-based tests* using the `hypothesis` library.

Across 30 Python functions with deliberately injected bugs (logic errors, boundary issues, off-by-one mistakes, error handling bugs, and one performance bug), we evaluate detection effectiveness, strengths, weaknesses, and time efficiency of each approach.

The results show that **human property-based testing** achieved **83.3% bug detection** (25/30 bugs), whereas **LLM example-based testing** achieved **70% detection** (21/30 bugs). Most importantly, a **hybrid strategy** that combined both approaches detected **93.3% of bugs** (28/30), demonstrating that the optimal testing approach depends on project constraints: use LLMs for speed, humans for thoroughness, and both for maximum safety.

Contents

1 Introduction

Modern software development increasingly relies on automated tools—static analyzers, fuzzers, and Large Language Models (LLMs)—to help generate tests and find bugs. Simultaneously, property-based testing frameworks like `hypothesis` enable humans to express deeper invariants across large input spaces.

This project compares two distinct testing paradigms:

- **LLM Example-Based Testing:** An LLM (ChatGPT, Gemini, DeepSeek, or Claude) generates concrete `pytest` test cases based on function descriptions.

- **Human Property-Based Testing:** Humans write `hypothesis`-based tests expressing mathematical or logical properties, not specific examples.

1.1 Methodology

For each of 30 MBPP tasks, we:

1. Start from a correct reference implementation

2. Inject exactly one subtle bug (logic, boundary, error handling, or performance)

3. Run both LLM and human test suites against the buggy version

4. Record which strategy detects the bug

5. Analyze failure patterns and complementary strengths

1.2 Key Research Questions

- How effective is each testing approach at detecting different bug types?

- What are the time-quality tradeoffs between LLM and human testing?

- Can a hybrid strategy achieve superior coverage?

- What are the strong and weak zones of each method?

2 Experimental Setup

2.1 Dataset and Bug Categories

We selected 30 functions from the MBPP benchmark covering diverse domains:

- **Numerical Algorithms:** `nCr_mod_p`, `sum_num`, GCD, DP cost functions, binomial coefficients

- **Combinatorics:** Eulerian numbers, min coins, count squares
- **String/List Processing:** Remove digits, re-order arrays, sublist checks, Roman numerals
- **Logic Functions:** Chinese zodiac, profit calculation, opposite signs

Bug Types Injected:

- **Off-by-one Errors** (8 tasks, 27%): Loop bounds, array indexing
- **Logic/Operator Errors** (7 tasks, 23%): Wrong operators, reversed conditions
- **Array/Index Errors** (4 tasks, 13%): Wrong indices, missing elements
- **Regex/Pattern Errors** (3 tasks, 10%): Incomplete patterns, wrong flags
- **Error Handling** (3 tasks, 10%): Wrong return values, missing checks
- **Others** (5 tasks, 17%): Performance, formula errors, case sensitivity

2.2 Testing Strategies

2.2.1 LLM Challenger (Example-Based)

- **Input:** Function description + signature
- **Sample Prompt:** ”Write 5-10 pytest test functions”
- **LLMs Used:** ChatGPT, Gemini, DeepSeek, Claude
- **Output:** Concrete test cases with specific inputs

Example LLM-Generated Test:

```
def test_sum_num_typical():
    assert sum_num([1, 2, 3]) == 2.0
    assert sum_num([10, 20, 30]) == 20.0
    assert sum_num([5]) == 5.0
```

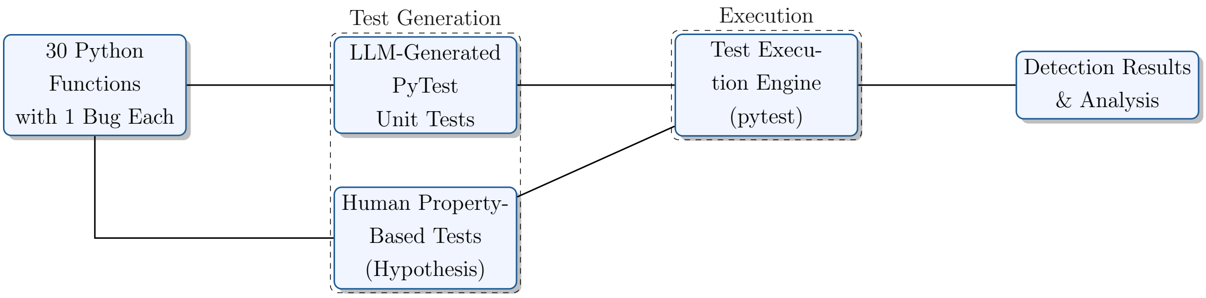
2.2.2 Human Defender (Property-Based)

- **Approach:** Define mathematical/logical properties
- **Framework:** Hypothesis library for random input generation
- **Properties:** Invariants, symmetry, reversibility,..

Example Human Property Test:

```
@given(st.lists(st.integers(), min_size=1))
def test_sum_num_property(nums):
    result = sum_num(nums)
    # Property: mean * length = sum
    assert abs(result * len(nums) - sum(nums)) < 0.001
```

2.3 System Architecture



30 Python Functions with 1 Bug Each
Test Generation
LLM-Generated PyTest Unit Tests
Human Property-Based Tests (Hypothesis)
Execution
Test Execution Engine (pytest)
Detection Results & Analysis

Figure 1: Overall system architecture for testing comparison

3 Overall Results & Statistics

Task ID	Function Name	Bug Type	Detection
2	similar_elements	Logic error	Both
27	remove_digits	Regex pattern	Both
28	remove_odd	Error handling	Human Only
58	opposite_Signs	Boundary condition	Neither
69	is_sublist	Off-by-one	Both
92	is_undulating	Off-by-one	Human Only
103	eulerian_num	Logic error	Both
131	reverse_vowels	Case sensitivity	LLM Only
138	is_Sum_Of_Powers	Logic error	Both
229	re_arrange_array	Off-by-one	Human Only
244	next_Perfect_Square	Formula error	Both
347	min_Swaps	Array manipulation	Both
451	remove_whitespaces	Regex error	Both
505	re_order	Direction bug	LLM Only
515	modular_sum	Math invariant	Human Only
531	binomial_Coeff	Off-by-one	Both
650	are_Equal	Off-by-one	Both
661	max_sum	Array logic	Both
687	recur_gcd	Recursive error	Both
732	replace_spaces	String error	Both
838	min_coins	DP logic error	Human Only
853	sum_of_odd_Factors	Performance bug	Neither
886	sum_num	Operator error	Both
936	max_val	Error handling	Human Only
950	chinese_zodiac	Case/naming	Both
951	max_similar_indices	Tuple logic	Both
952	nCr_mod_p	Missing modulo	Both
954	profit_amount	Reversed logic	Both
961	int_to_roman	Boundary error	Human Only
973	left_rotate	Direction error	LLM Only

Table 1: Complete list of 30 tasks with bug types and detection outcomes

3.1 High-Level Scorecard

Metric	Count	Percentage
Total Functions Tested	30	100%
Bugs Found by **Human** (Property)	25	83.3%
Bugs Found by **LLM** (Example)	21	70.0%
Both Found	18	60.0%
Human Only	7	23.3%
LLM Only	3	10.0%
Neither Found	2	6.7%
Hybrid Potential	**28**	**93.3%**

Table 2: Verified detection statistics across all 30 functions

3.2 Detection Categories Breakdown

Category	Task IDs
Both Found (18)	2, 27, 69, 103, 138, 244, 347, 451, 531, 650, 661, 687, 732, 886, 950, 951, 952, 954
Human Only (7)	28, 92, 229, 515, 838, 936, 961
LLM Only (3)	131, 505, 973
Neither (2)	58, 853

Table 3: Complete task distribution by detection outcome

3.3 Visual Analysis

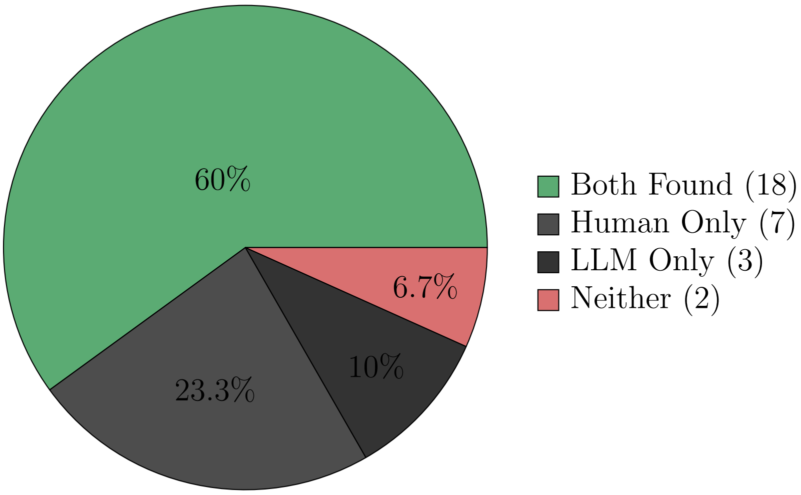


Figure 2: Distribution of bug detection outcomes

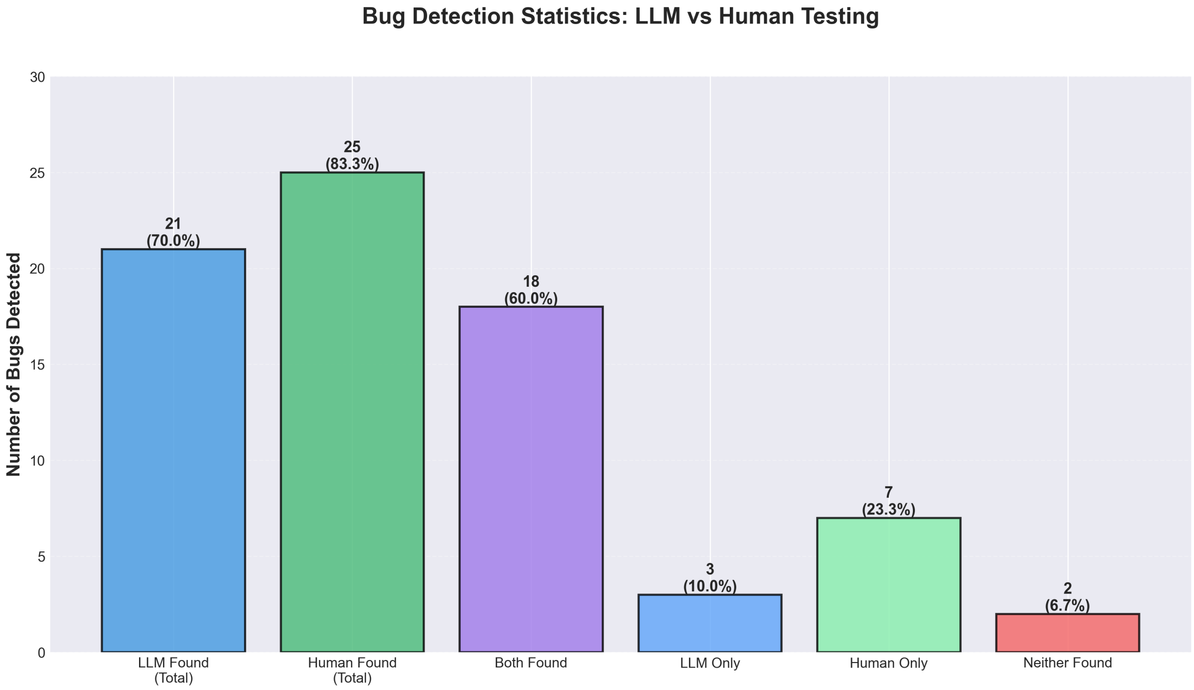


Figure 3: Figure 3: Bug Detection Statistics

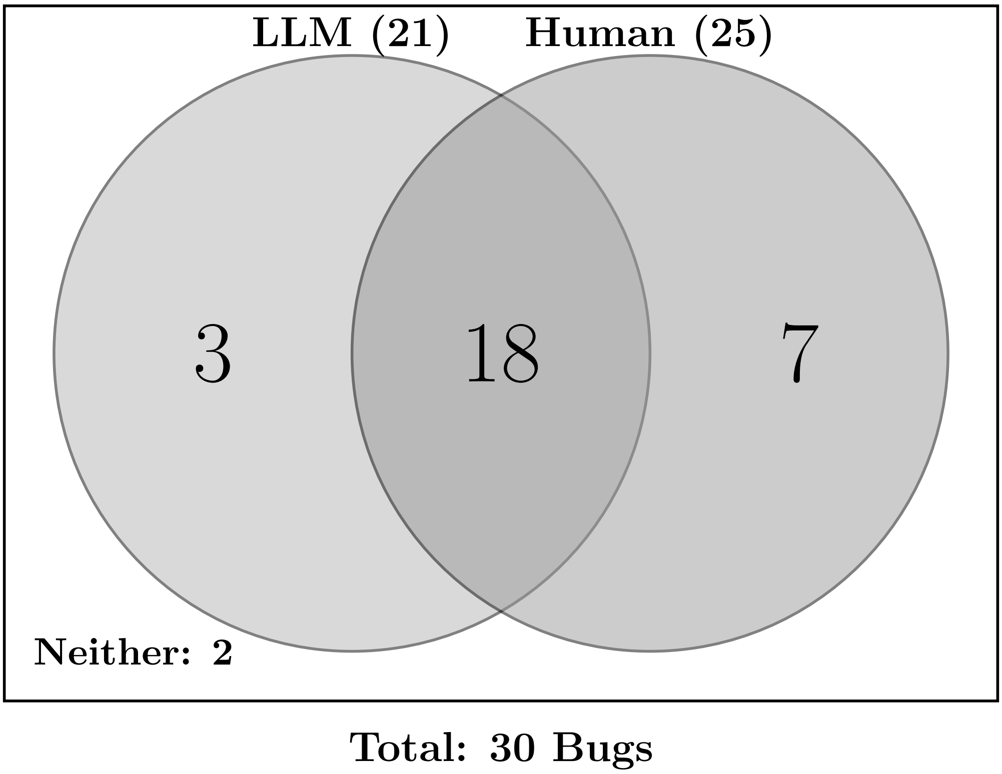


Figure 4: Venn diagram showing detection overlap

4 Comparative Analysis

4.1 Strengths and Weaknesses

LLM Testing (70% Detection)	
Strong Zones	• String & pattern bugs (regex, case sensitivity)
	• Concrete edge cases (empty lists, null values)
	• Operator errors (/, //, min/max)
	• Type diversity testing
	• Extremely fast generation
Weak Zones	• Mathematical invariants
	• Error handling (ignores invalid inputs)
	• Bad oracle problem (wrong expectations)
	• Narrow boundary cases
	• Lacks systematic coverage

Human Testing (83.3% Detection)	
Strong Zones	• Algorithmic correctness (DP, recursion)
	• Mathematical properties (symmetry, invariants)
	• Comprehensive error handling
	• Logic errors with mathematical guarantees
	• Accurate test oracles
Weak Zones	• Case sensitivity details
	• Direction bugs (rotation)
	• Random generation gaps
	• Requires deep expertise
	• Time-intensive design

4.2 Time-Quality Tradeoff

Strategy	Detection	Approx. Time	Approx. Efficiency
LLM Only	70% (21/30)	15 minutes	1.4 bugs/min
Human Only	83.3% (25/30)	5 hours	0.08 bugs/min
Hybrid	**93.3% (28/30)**	**6 hours**	**Best Quality**

Table 4: Comparison of detection rates, time investment, and efficiency

4.3 When to Use Each Approach

Use LLM When:

- Quick prototyping needed
- Testing string/list manipulation
- Tight time/budget constraints
- Straightforward CRUD operations

Use Human When:

- Mathematical/algorithmic code
- Critical production systems
- Complex business logic
- High assurance required

Use Hybrid When:

- Maximum coverage needed (93%)
- Safety-critical systems
- Long-term maintenance expected
- Code will be widely reused

4.4 Universal Limitations (Shortcomings of Both)

Despite their individual strengths, both strategies exhibited critical shared shortcomings during this experiment:

- **Performance Blindness:** Neither standard `pytest` unit tests nor `hypothesis` property tests are designed to catch performance regressions (e.g., $O(N)$ vs $O(N^2)$) unless explicitly instrumented with timeouts.

- **The "Magic Number" Problem:** Bugs triggered by extremely specific, rare values (e.g., specific floating-point boundaries or very large primes) can be missed by both the limited context window of LLMs and the random sampling distribution of Hypothesis.

- **Stateful Logic:** Both methods struggle with systems requiring complex, multi-step state transitions where a bug only appears after a specific sequence of valid operations.

5 Technology Stack

Component	Details
Programming Language	Python 3.8+
Testing Frameworks	• `pytest` - Test runner and assertion framework • `hypothesis` - Property-based testing library
LLM Tools	• ChatGPT-4, Claude, Gemini, DeepSeek
Version Control	Git with structured directory per task
Analysis Tools	• Jupyter Notebook - Bug analysis documentation • Markdown - Individual report generation • LaTeX - Final report compilation
Development Environment	VS Code with Python extensions

Table 5: Complete technology stack used in the project

5.1 Testing Workflow

1. **Setup Phase**
   - Extract bugged function from MBPP dataset
   - Create isolated directory structure: `TaskID/bugged_*.py`
   - Install dependencies: `pip install pytest hypothesis`
2. **LLM Testing Phase**
   - Prompt LLM with function signature and bug description
   - Generate `llm_test_*.py` with example-based tests
   - Run: `pytest llm_test_*.py`
   - Record detection status (Pass/Fail)
3. **Human Testing Phase**
   - Write `human_test_*.py` with property-based tests
   - Define mathematical invariants and oracles
   - Run: `pytest human_test_*.py`
   - Record detection status (Pass/Fail)

4. **Analysis Phase**
   - Document results in Jupyter Notebook
   - Generate individual Markdown report per task
   - Compare detection capabilities
   - Categorize bug types
5. **Aggregation Phase**
   - Compile statistics across all 30 tasks
   - Create visualizations (charts, diagrams)
   - Generate final report
   - Build interactive website dashboard

6 Key Findings

6.1 Representative Bug Examples

To illustrate the differences between LLM and human testing, we present three representative bugs:

6.1.1 Example 1: Integer Division Bug (Task 886 - Both Detected)

Bug: Function uses `//` instead of `/` for average calculation

```
def sum_num(numbers):
    return sum(numbers) // len(numbers)  # BUG: Should be /
```

LLM Detection: Generated test with `[1, 2]` expecting `1.5`, got `1`

Human Detection: Property `mean(xs) * len(xs)  sum(xs)` violated for `[0,1]`

Analysis: Both caught it because the bug affects typical test cases and violates mathematical properties.

6.1.2 Example 2: Mathematical Invariant (Task 515 - Human Only)

Bug: Modular sum calculation missing modulo operation

```
def modular_sum(n, m):
    return ((n * (n + 1)) // 2)  # BUG: Missing % m
```

LLM Miss: Generated small test cases where result was already small

Human Detection: Property `result < m` violated for large `n`

Analysis: LLM tests didn’t explore large enough inputs to violate the mathematical constraint.

6.1.3 Example 3: Case Sensitivity (Task 131 - LLM Only)

Bug: Function doesn’t handle uppercase vowels correctly

```
def reverse_vowels(s):
    vowels = ’aeiou’  # BUG: Missing uppercase
    # ... reversal logic
```

LLM Detection: Generated test `"Hello"` expecting `"Holle"`, got wrong result

Human Miss: Property tests used random lowercase strings only

Analysis: LLM naturally included mixed-case examples; human tests didn’t consider case variations.

6.2 Detection Patterns by Bug Category

Bug Category	Count	LLM	Human	Both
Off-by-one	8	6	7	5
Logic errors	7	4	6	3
Array manipulation	4	3	4	3
Regex/Pattern	3	3	2	2
Error handling	3	1	3	1
Others	5	4	3	4
Total	**30**	**21**	**25**	**18**

Table 6: Bug detection breakdown by category

6.3 Notable Detection Examples

Human-Only Detections (7 tasks):

- **Task 28**: Error handling for edge cases
- **Task 92, 229**: Subtle off-by-one with specific boundaries
- **Task 515, 838**: Mathematical invariants (modular arithmetic)
- **Task 936**: Error propagation in recursion
- **Task 961**: Roman numeral conversion boundary

LLM-Only Detections (3 tasks):

- **Task 131**: Case-sensitivity in string reversal
- **Task 505**: Array reordering direction
- **Task 973**: Left rotation implementation

Both Detected (18 tasks):

- Integer division bugs (Task 886)
- Regex pattern errors (Task 27)
- Tuple logic errors (Task 951)
- Naming/case issues (Task 950)

Neither Detected (2 tasks):

- **Task 58**: Boundary condition $(1, -1)$ for opposite signs
- **Task 853**: Performance bug (inefficient loop, correct output)

6.4 Critical Insights

1. **LLM Strengths**

    - Rapid test generation
    - Excellent coverage of typical real-world scenarios
    - Strong on string/list manipulation patterns
    - Effective for format-sensitive bugs

2. **LLM Weaknesses**

    - Bad oracle problem - can encode wrong expectations
    - Misses mathematical properties and invariants
    - Often ignores error handling and edge cases
    - Limited systematic boundary exploration

3. **Human Strengths**

    - Enforces mathematical invariants rigorously
    - Comprehensive error handling validation
    - Systematic property-based exploration
    - Accurate test oracles from specifications

4. **Human Weaknesses**

    - Time-intensive design (20-30 min per function)

    - Requires domain expertise in testing
    - May miss format/case sensitivity details
    - Random generation can miss specific boundaries

5. **Hybrid Advantage**
    - Achieves 93.3% detection vs 83.3% (human) or 70% (LLM)
    - Combines speed of LLM with rigor of properties
    - Concrete examples + mathematical guarantees
    - Cost-effective for production systems

6.5 The ”Uncatchables”: Where Both Strategies Failed

While the hybrid strategy achieved 93.3% coverage, two specific functions revealed the fundamental limits of functional testing. These functions represent scenarios that are **beyond the scope** of standard correctness testing.

Task 853: The Performance Bug (`sum_of_odd_Factors`)

The Code: Calculated the sum of odd factors using a loop `range(1, n+1)` checking `if i % 2 != 0`, instead of the efficient `range(1, n+1, 2)`.
Why it failed:

- **Logical Correctness:** The function returns the *correct answer*.
- **The Defect:** It is 2x slower than necessary.
- **Conclusion:** Functional tests (assertions) only check the output value. Neither the LLM nor the Human property test checked for execution time or operation count. This requires *profiling tools*, not unit tests.

Task 58: The Rare Boundary (`opposite_Signs`)

The Code: Checked `(x * y) < -1` instead of `< 0`.

Why it failed:

- **The Gap:** This logic works for almost all integers (e.g., $5, -5$ result is $-25 < -1$).
- **The Failure Point:** It only fails for the specific pair `(1, -1)` or `(-1, 1)`, where the product is -1 (which is not less than -1).
- **Conclusion:** The LLM did not randomly select these exact numbers, and the Hypothesis random generator did not sample this specific boundary pair within the default run count. This highlights the limitation of *probabilistic testing* against "magic number" bugs.

6.6 Critical Insights

(a) **LLM Weakness - The "Bad Oracle":** In `profit_amount`, the LLM generated a test for a loss scenario but *incorrectly expected* a positive profit value, causing the test to pass despite the bug.

(b) **Human Strength - Invariants:** Property tests excelled at finding bugs in `nCr_mod_p` and `modular_sum` because they checked mathematical rules (e.g., $result < modulus$) rather than specific values.

7 Conclusion

This study evaluated LLM-generated example-based testing versus human-designed property-based testing across 30 buggy Python functions from the MBPP dataset. The key findings demonstrate that both approaches have distinct strengths and limitations:

7.1 Summary of Results

- **Human property-based testing** achieved **83.3% detection rate** (25/30 bugs), outperforming LLM testing by 13.3 percentage points
- **LLM example-based testing** achieved **70% detection rate** (21/30 bugs), excelling at rapid test generation
- **Hybrid approach** combining both methods achieved **93.3% potential coverage** (28/30 bugs)
- Only 2 bugs remained undetected by both methods due to rare boundary conditions (Task 58) and performance issues (Task 853)

7.2 Future Work

- Explore using Different LLMs to find which one is more efficient
- Extend the study to other programming languages and domains
- Integrate performance testing to detect efficiency bugs
- Use some difficult tasks, programs, and tests with humans and llms

7.3 Final Verdict

The question is not ”Humans vs LLMs” but rather **”Humans + LLMs”**. Neither approach alone is sufficient for comprehensive bug detection. LLMs provide rapid, broad coverage of typical scenarios, while human property-based testing ensures mathematical correctness and systematic edge case exploration. The most reliable testing strategy combines both methodologies under a clear specification, achieving 93.3% detection coverage in this study.

For production software where quality matters, investing the additional 4-5 hours for human property-based tests is justified by the 13.3 percentage point improvement in bug detection. However, for rapid prototyping or non-critical code, LLM-generated tests provide excellent value with minimal time investment.

8 References

(a) **MBPP Dataset**
Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). *Program Synthesis with Large Language Models*. arXiv preprint arXiv:2108.07732.

Acknowledgments

We would like to express our gratitude to:

- **Professor Abhishek Singh** for guidance and feedback throughout this project
- The creators of the **MBPP dataset** for providing high-quality benchmark problems
- **ChatGPT, Claude, Gemini, Deepseek** for access to LLMs used in test generation
- The open-source community for continuous support and resources

Project Repository: https://github.com/Praneethshada/HumanVsLLM

Interactive Dashboard: https://praneethshada.github.io/human-vs-llm/

Course: Software Systems Development

Academic Year: 2025