

University of Sheffield

Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs

Ziyu Li and Donghwan Shin

15 April 2024 | CAIN 2024

LLMs in Software Engineering

arXiv:2308.10620v3 [cs.SE] 3 Sep 2023

Large Language Models for Software Engineering: A Systematic Literature Review

XINYI HOU*, Huazhong University of Science and Technology, China
YANJIE ZHAO*, Monash University, Australia
YUE LIU, Monash University, Australia
ZHOU YANG, Singapore Management University, Singapore
KAILONG WANG, Huazhong University of Science and Technology, China
LI LI, Beihang University, China
XIAPU LUO, The Hong Kong Polytechnic University, China
DAVID LO, Singapore Management University, Singapore
JOHN GRUNDY, Monash University, Australia
HAOYU WANG†, Huazhong University of Science and Technology, China

Large Language Models (LLMs) have significantly impacted numerous domains, including Software Engineering (SE). Many recent publications have explored LLMs applied to various SE tasks. Nevertheless, a comprehensive understanding of the application, effects, and possible limitations of LLMs on SE is still in its early stages. To bridge this gap, we conducted a systematic literature review on LLM4SE, with a particular focus on understanding how LLMs can be exploited to optimize processes and outcomes. We collect and analyze 229 research papers from 2017 to 2023 to answer four key research questions (RQs). In RQ1, we categorize different LLMs that have been employed in SE tasks, characterizing their distinctive features and uses. In RQ2, we analyze the methods used in data collection, preprocessing, and application highlighting the role of well-curated datasets for successful LLM for SE implementation. RQ3 investigates the strategies employed to optimize and evaluate the performance of LLMs in SE. Finally, RQ4 examines the specific SE tasks where LLMs have shown success to date, illustrating their practical contributions to the field. From the answers to these RQs, we discuss the current state-of-the-art and trends, identifying gaps in existing research, and flagging promising areas for future study.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Software and its engineering** → **Software development techniques**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: Software Engineering, Large Language Model, Survey

*Co-first authors who contributed equally to this work.
†Haoyu Wang is the corresponding author (haoyuwang@hust.edu.cn).

Authors' addresses: Xinyi Hou, xinyihou@hust.edu.cn, Huazhong University of Science and Technology, Wuhan, China; Yanjie Zhao, Yanjie.Zhao@monash.edu, Monash University, Melbourne, Australia; Yue Liu, yue.liu1@monash.edu, Monash University, Melbourne, Australia; Zhou Yang, zyang@smu.edu.sg, Singapore Management University, Singapore; Kailong

arXiv:2310.03533v2 [cs.SE] 8 Oct 2023

Large Language Models for Software Engineering: Survey and Open Problems

Angela Fan
Generative AI Team
Meta Platforms Inc.
New York, NY, USA

Beliz Gokkaya
PyTorch Team
Meta Platforms Inc.
Menlo Park, CA, USA

Mark Harman
Instagram Product Foundation
Meta Platforms Inc.
London, UK

Mitya Lyubarskiy
Developer Infrastructure
Meta Platforms Inc.
London, UK

Shubho Sengupta
FAIR
Meta Platforms Inc.
Menlo Park, CA, USA

Shin Yoo
School of Computing
KAIST
Daejeon, Korea

Jie M. Zhang
Department of Informatics
King's College London
London, UK

***Abstract*—This paper provides a survey of the emerging area of Large Language Models (LLMs) for Software Engineering (SE). It also sets out open research challenges for the application of LLMs to technical problems faced by software engineers. LLMs' emergent properties bring novelty and creativity with applications right across the spectrum of Software Engineering activities including coding, design, requirements, repair, refactoring, performance improvement, documentation and analytics. However, these very same emergent properties also pose significant technical challenges; we need techniques that can reliably weed out incorrect solutions, such as hallucinations. Our survey reveals the pivotal role that hybrid techniques (traditional SE plus LLMs) have to play in the development and deployment of reliable, efficient and effective LLM-based SE.**

***Index Terms*—Automated Program Repair, Design, Documentation generation, Generative AI, Genetic Improvement, Human-Computer Interaction, Large Language Models, Refactoring, Requirements engineering, Search Based Software Engineering (SBSE), Software Analytics, Software Engineering Education, Software Processes, Software Maintenance and Evolution, Software Testing.**

I. Introduction

This paper surveys the recent developments, advances and empirical results on LLM-based SE; the application of Large Language Models (LLMs) to Software Engineering (SE) applications. We use the survey to highlight gaps in this rapidly developing, but as yet embryonic, research literature. Based on gaps in the literature and technical opportunities, we also identify open problems and challenges for the software engineering research community.

While any survey of such a rapidly expanding area can trends and well-established approaches and subdisciplines within Software Engineering.

Although, overall, we find considerable grounds for optimism, there remain important technical challenges, which are likely to inform the research agenda for several years. Many authors have highlighted, both scientifically and anecdotally, that hallucination is a pervasive problem for LLMs [1] and also that it poses specific problems for LLM-based SE [2]. As with human intelligence, hallucination means that the LLM can create fictitious output. In the context of software engineering, it means that the engineering artefacts created could be incorrect, yet appear plausible; LLMs may introduce bugs.

However, unlike many other applications of LLMs, software engineers are typically blessed with automatable ground truth (software execution), against which most software engineering artefacts can be evaluated. Also, the software engineering research community has already devoted a great deal of time to producing automated and semi-automated techniques for checking the potentially incorrect results produced by humans. This means that, for the discipline and the research community, there is a great deal of experience and expertise on which to draw, when tackling the challenges posed by issues like hallucination.

Clearly, automated testing techniques [3]–[5] will have a central role to play in ensuring correctness, just as they already do for human-engineered artefacts. When generating entirely new features and systems, automated test data generation suffers from the lack of an automatable oracle [6] (an au-

“How can we test the programming capabilities of LLMs?”

LLM Testing using Benchmarks

LLM Testing using Benchmarks

arXiv:2107.03374v2 [cs.LG] 14 Jul 2021

Evaluating Large Language Models Trained on Code

Mark Chen [*1] Jerry Tworek [*1] Heewoo Jun [*1] Qiming Yuan [*1] Henrique Ponde de Oliveira Pinto [*1]
Jared Kaplan [*2] Harri Edwards [1] Yuri Burda [1] Nicholas Joseph [2] Greg Brockman [1] Alex Ray [1] Raul Puri [1]
Gretchen Krueger [1] Michael Petrov [1] Heidy Khlaaf [3] Girish Sastry [1] Pamela Mishkin [1] Brooke Chan [1]
Scott Gray [1] Nick Ryder [1] Mikhail Pavlov [1] Alethea Power [1] Lukasz Kaiser [1] Mohammad Bavarian [1]
Clemens Winter [1] Philippe Tillet [1] Felipe Petroski Such [1] Dave Cummings [1] Matthias Plappert [1]
Fotios Chantzis [1] Elizabeth Barnes [1] Ariel Herbert-Voss [1] William Hebgen Guss [1] Alex Nichol [1] Alex Paino [1]
Nikolas Tezak [1] Jie Tang [1] Igor Babuschkin [1] Suchir Balaji [1] Shantanu Jain [1] William Saunders [1]
Christopher Hesse [1] Andrew N. Carr [1] Jan Leike [1] Josh Achiam [1] Vedant Misra [1] Evan Morikawa [1]
Alec Radford [1] Matthew Knight [1] Miles Brundage [1] Mira Murati [1] Katie Mayer [1] Peter Welinder [1]
Bob McGrew [1] Dario Amodei [2] Sam McCandlish [2] Ilya Sutskever [1] Wojciech Zaremba [1]

Abstract

We introduce Codex, a GPT language model fine-tuned on publicly available code from GitHub, and study its Python code-writing capabilities. A distinct production version of Codex powers GitHub Copilot. On HumanEval, a new evaluation set we release to measure functional correctness for synthesizing programs from docstrings, our model solves 28.8% of the problems, while GPT-3 solves 0% and GPT-J solves 11.4%. Furthermore, we find that repeated sampling from the model is a surprisingly effective strategy for producing working solutions to difficult prompts. Using this method, we solve 70.2% of our problems with 100 samples per problem. Careful investigation of our model reveals its limitations, including difficulty with docstrings describing long chains of operations and with binding operations to variables. Finally, we discuss the potential broader impacts of deploying powerful code generation technologies, covering safety, security, and economics.

1. Introduction

Scalable sequence prediction models (Graves, 2014; Vaswani et al., 2017; Child et al., 2019) have become a general-purpose method for generation and representation learning in many domains, including natural language processing (Mikolov et al., 2013; Sutskever et al., 2014; Dai & Le, 2015; Peters et al., 2018; Radford et al., 2018; Devlin et al., 2018), computer vision (Van Oord et al., 2016; Menick & Kalchbrenner, 2018; Chen et al., 2020; Bao et al., 2021), audio and speech processing (Oord et al., 2016; 2018; Dhariwal et al., 2020; Baevski et al., 2020), biology (Alley et al., 2019; Rives et al., 2021), and even across multiple modalities (Das et al., 2017; Lu et al., 2019; Ramesh et al., 2021; Zellers et al., 2021). More recently, language models have also fueled progress towards the longstanding challenge of program synthesis (Simon, 1963; Manna & Waldinger, 1971), spurred by the presence of code in large datasets (Husain et al., 2019; Gao et al., 2020) and the resulting programming capabilities of language models trained on these datasets (Wang & Komatsuzaki, 2021). Popular language modeling objectives like masked language modeling (Devlin et al., 2018) and span prediction (Raffel et al., 2020) have also been adapted to train their programming counterparts CodeBERT (Feng et al., 2020) and PyMT5 (Clement et al., 2020).

Similarly, our early investigation of GPT-3 (Brown et al., 2020) revealed that it could generate simple programs from Python docstrings. While rudimentary, this capability was exciting because GPT-3 was not explicitly trained for code generation. Given the considerable success of large language models in other modalities and the abundance of publicly available code, we hypothesized that a specialized GPT model, called Codex, could excel at a variety of coding tasks. This paper describes several early Codex models, whose descendants power GitHub Copilot and the Codex models in the OpenAI API.

[*]Equal contribution
[1]OpenAI, San Francisco, California, USA.
[2]Anthropic AI, San Francisco, California, USA. Work performed while at OpenAI.
[3]Zipline, South San Francisco, California, USA. Work performed while at OpenAI.
Correspondence to: Mark Chen <mark@openai.com>, Jerry Tworek <jt@openai.com>, Heewoo Jun <heewoo@openai.com>, Qiming Yuan <qiming@openai.com>.

LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count
prompt:

```
def even_odd_count(num):
    """Given an integer. return a tuple that has
        the number of even and odd digits respectively.

     Example:
        even_odd_count(-12) ==> (1, 1)
        even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]

LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count
prompt:

```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]



LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count
prompt:

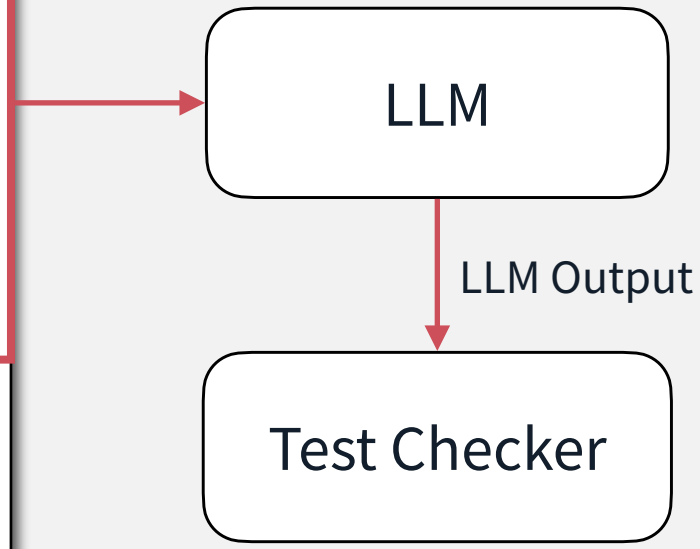
```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]



LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count

prompt:

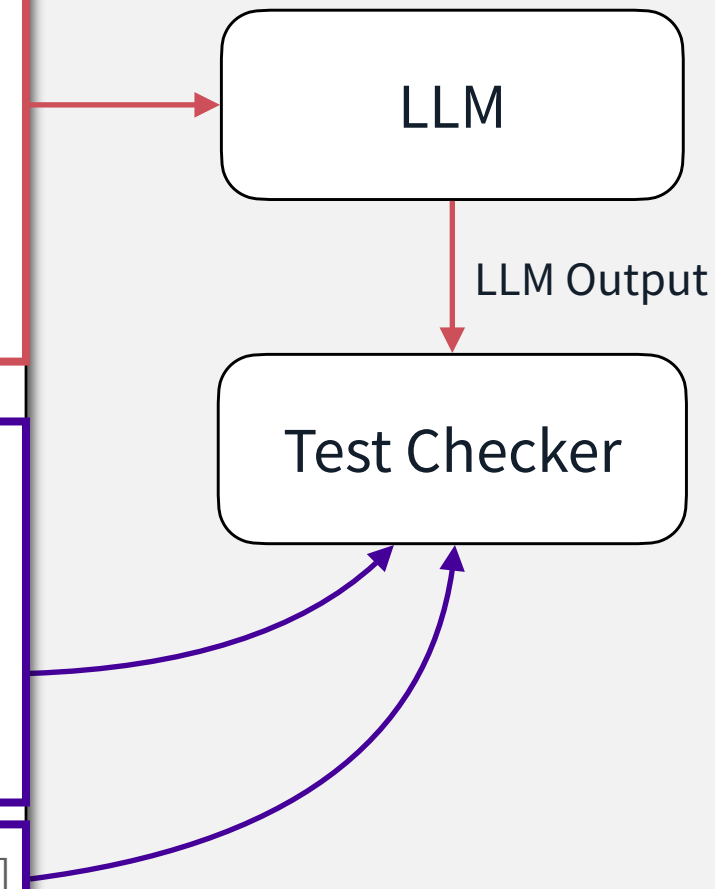
```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]



LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count

prompt:

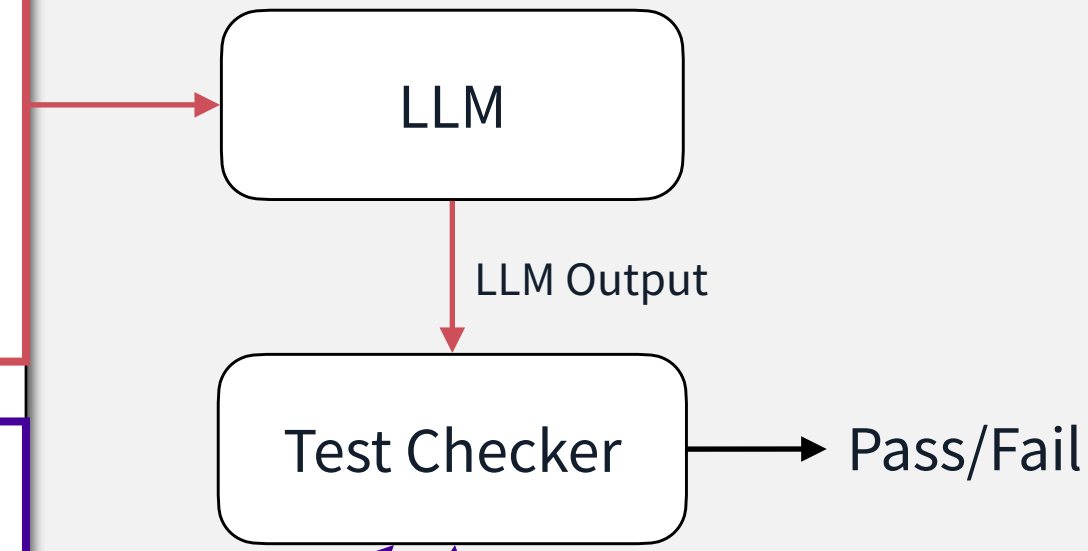
```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]



LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count

prompt:

```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

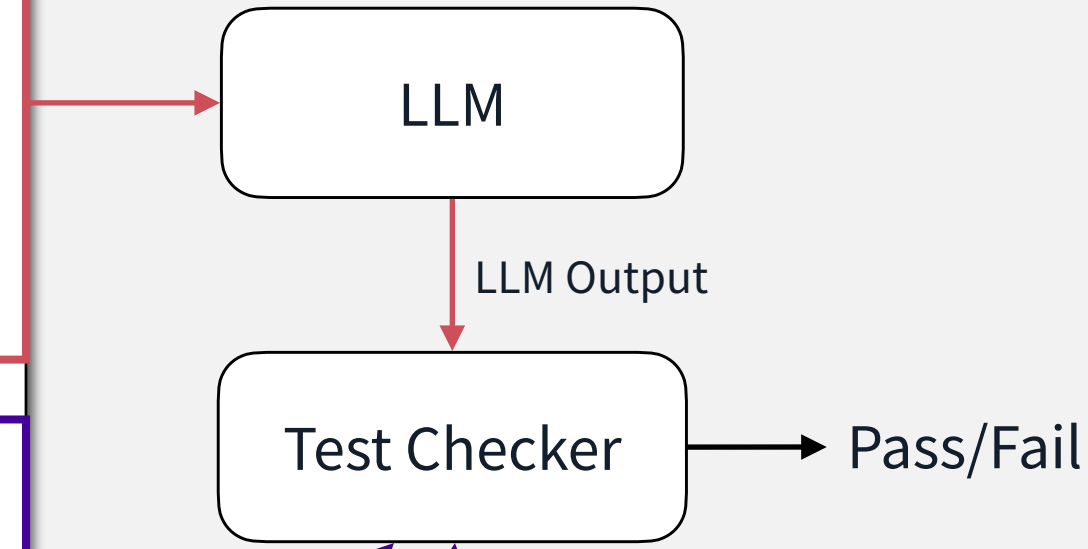
     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]

- To measure **functional** correctness
- Code **writing**, not code reading



“Can LLMs understand code fragments?”

“Can LLMs understand code fragments?”

“Can LLMs identify subtle inconsistencies between code fragments and their descriptions?”

Key Idea

Key Idea

We can reuse the **canonical solution** (code fragment) and the **prompt** (task description) in the existing LLM testing benchmarks (e.g., HumanEval).

By design, the code and its description are **consistent**.

Key Idea

We can reuse the **canonical solution** (code fragment) and the **prompt** (task description) in the existing LLM testing benchmarks (e.g., HumanEval).

- By design, the code and its description are **consistent**.

We can apply **mutations** to the canonical solution to create **subtle inconsistencies** between the code and its description.

- Mutation analysis, including various **mutation operators**, has been widely studied to generate subtle mutants systematically.

Key Idea

We can reuse the **canonical solution** (code fragment) and the **prompt** (task description) in the existing LLM testing benchmarks (e.g., HumanEval).

By design, the code and its description are **consistent**.

We can apply **mutations** to the canonical solution to create **subtle inconsistencies** between the code and its description.

Mutation analysis, including various **mutation operators**, has been widely studied to generate subtle mutants systematically.

Mutation Operators

Name	Description	Example
AOR	Arithmetic Operator Replacement	a + b → a − b
LVR	Literal Value Replacement	10 → 9
ROR	Relational Operator Replacement	a < b → a >= b
STD	Statement Deletion	remove one line

LLM Testing using Benchmarks

task_id: HumanEval/155
entry_point: even_odd_count

prompt:

```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

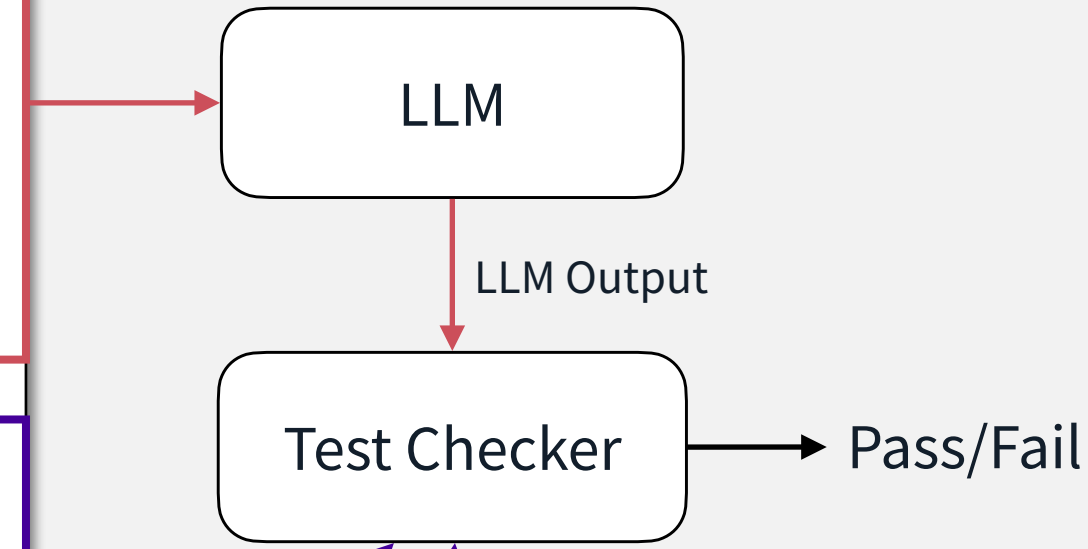
     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]

- To measure **functional** correctness
- Code **writing**, not code reading



Overview: Mutation-based Consistency Testing (MCT) for LLMs

task_id: HumanEval/155
entry_point: even_odd_count

prompt:

```
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

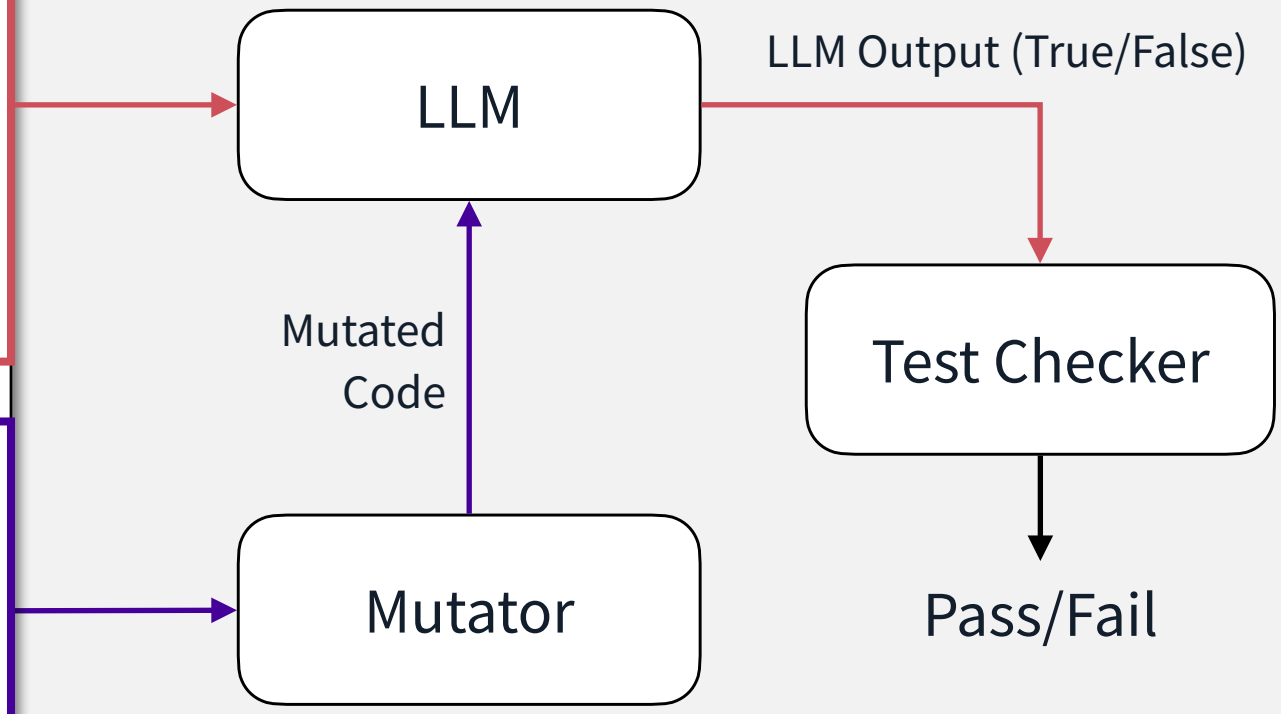
     Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
```

canonical_solution:

```
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
```

base_input: [[7], [-78], [3452], [346211], [-345821], …]

- To measure **consistency checking capability**
- Focusing on code **reading**



MCT in Pseudo-Code

Algorithm 1: Mutation-based Consistency Testing (MCT) for LLMs

Input : LLM under Test $\mathcal{L}$,
Set of Consistent Pairs $D = \{(des, imp), \dots\}$,
Set of Mutation Operators U

Output : MCT Score s

Set of Passed Mutated Data $P \leftarrow \emptyset$
Set of Failed Mutated Data $F \leftarrow \emptyset$
foreach *Consistent Pair* $(des, imp) \in D$ **do**
  if `isConsistent`$(\mathcal{L}, des, imp)$ **then**
    Set of Mutants $M \leftarrow$ `genMutants`(imp, U)
    foreach *Mutant* $m \in M$ **do**
      if $\neg$`isConsistent`$(\mathcal{L}, desc, m)$ **then**
        $P \leftarrow P \cup \{(des, m)\}$
      else
        $F \leftarrow F \cup \{(des, m)\}$
MCT Score $s \leftarrow \frac{|P|}{|P|+|F|} \times 100$
return s

MCT in Pseudo-Code

Algorithm 1: Mutation-based Consistency Testing (MCT) for LLMs

Input : LLM under Test $\mathcal{L}$,
Set of Consistent Pairs $D = \{(des, imp), \dots\}$,
Set of Mutation Operators U

Output : MCT Score s

Set of Passed Mutated Data $P \leftarrow \emptyset$
Set of Failed Mutated Data $F \leftarrow \emptyset$
foreach *Consistent Pair* $(des, imp) \in D$ **do**
  if `isConsistent`$(\mathcal{L}, des, imp)$ **then**
    Set of Mutants $M \leftarrow$ `genMutants`(imp, U)
    foreach *Mutant* $m \in M$ **do**
      if $\neg$`isConsistent`$(\mathcal{L}, desc, m)$ **then**
        $P \leftarrow P \cup \{(des, m)\}$
      else
        $F \leftarrow F \cup \{(des, m)\}$
MCT Score $s \leftarrow \frac{|P|}{|P|+|F|} \times 100$
return s

Pairs of descriptions (*des*) and their implementations (*imp*)

MCT in Pseudo-Code

Algorithm 1: Mutation-based Consistency Testing (MCT) for LLMs

Input : LLM under Test $\mathcal{L}$,
Set of Consistent Pairs $D = \{(des, imp), \dots\}$,
Set of Mutation Operators U

Output : MCT Score s

Set of Passed Mutated Data $P \leftarrow \emptyset$
Set of Failed Mutated Data $F \leftarrow \emptyset$
foreach *Consistent Pair* $(des, imp) \in D$ **do**
  if `isConsistent`$(\mathcal{L}, des, imp)$ **then**
    Set of Mutants $M \leftarrow$ `genMutants`(imp, U)
    foreach *Mutant* $m \in M$ **do**
      if $\neg$`isConsistent`$(\mathcal{L}, desc, m)$ **then**
        $P \leftarrow P \cup \{(des, m)\}$
      else
        $F \leftarrow F \cup \{(des, m)\}$
MCT Score $s \leftarrow \frac{|P|}{|P|+|F|} \times 100$
return s

Pairs of descriptions (*des*) and their implementations (*imp*)

To consider correctly identified *des-imp* pairs only

MCT in Pseudo-Code

Algorithm 1: Mutation-based Consistency Testing (MCT) for LLMs

Input : LLM under Test $\mathcal{L}$,
Set of Consistent Pairs $D = \{(des, imp), \dots\}$,
Set of Mutation Operators U

Output : MCT Score s

Set of Passed Mutated Data $P \leftarrow \emptyset$
Set of Failed Mutated Data $F \leftarrow \emptyset$
foreach *Consistent Pair* $(des, imp) \in D$ **do**
  if `isConsistent`$(\mathcal{L}, des, imp)$ **then**
    Set of Mutants $M \leftarrow$ `genMutants`(imp, U)
    foreach *Mutant* $m \in M$ **do**
      if $\neg$`isConsistent`$(\mathcal{L}, desc, m)$ **then**
        $P \leftarrow P \cup \{(des, m)\}$
      else
        $F \leftarrow F \cup \{(des, m)\}$
MCT Score $s \leftarrow \frac{|P|}{|P|+|F|} \times 100$
return s

Pairs of descriptions (*des*) and their implementations (*imp*)

To consider correctly identified *des-imp* pairs only

The description and its mutated implementation must be inconsistent by definition

Prompting (base)

We carefully crafted a prompt considering three essential aspects:

(1) clear task definition

(2) structured input

(3) explicit evaluation criteria

```
I'm presenting you with a program
description and its corresponding code.
Description: {DESCRIPTION HERE}
Code: {CODE HERE}
Evaluate the code based on the description.
Respond with:
1 - If the code functionally and syntactically
    matches the description completely without any
     minor inconsistency.
2 - Otherwise.
Please respond only with a single number:
'1' or '2'.
```

One-Shot Prompting (optional)

Provides an **example** to guide the LLM's response.

Such few-shot prompting helps the LLM better understand the context and requirements.

The figure on the right is the example part to be **appended at the end** of the base prompt.

The same example is used for all prompts.

Here is an example for you. When given this pair:

Description:

```
from typing import List\n\n\ndef
    has_close_elements(numbers: List[float],
    threshold: float) -> bool:
    """ Check if in given list of numbers, are any
     two numbers closer to each other than given
    threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0,
    5.0, 2.0], 0.3)
    True
    """
```

Code:

```
for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem * elem2)
                if distance < threshold:
                    return True

    return False
```

The code did not match the description, the code should be:

```
for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```

So the correct answer is: 2

Case Study Design

RQ1. How do different LLMs fare in terms of MCT for different **mutation operators**?

RQ2. How do different LLMs fare in terms of MCT for different **programming languages**?

RQ3. Can we explain the MCT results of LLMs in terms of **input characteristics**?

RQ4. How do the MCT results of LLMs vary with **zero-shot** and **one-shot promptings**?

Case Study Design

RQ1. How do different LLMs fare in terms of MCT for different **mutation operators**?

RQ2. How do different LLMs fare in terms of MCT for different **programming languages**?

RQ3. Can we explain the MCT results of LLMs in terms of **input characteristics**?

RQ4. How do the MCT results of LLMs vary with **zero-shot** and **one-shot promptings**?

HumanEval-X

164 Programming tasks for six languages
(Python, C++, Java, Go, JavaScript (JS), and Rust)

Case Study Design

RQ1. How do different LLMs fare in terms of MCT for different **mutation operators**?

RQ2. How do different LLMs fare in terms of MCT for different **programming languages**?

RQ3. Can we explain the MCT results of LLMs in terms of **input characteristics**?

RQ4. How do the MCT results of LLMs vary with **zero-shot** and **one-shot promptings**?

HumanEval-X

164 Programming tasks for six languages
(Python, C++, Java, Go, JavaScript (JS), and Rust)

GPT-3.5 & GPT-4

Two most popular LLMs from OpenAI

Case Study Design

RQ1. How do different LLMs fare in terms of MCT for different **mutation operators**?

RQ2. How do different LLMs fare in terms of MCT for different **programming languages**?

RQ3. Can we explain the MCT results of LLMs in terms of **input characteristics**?

RQ4. How do the MCT results of LLMs vary with **zero-shot** and **one-shot promptings**?

HumanEval-X

164 Programming tasks for six languages
(Python, C++, Java, Go, JavaScript (JS), and Rust)

GPT-3.5 & GPT-4

Two most popular LLMs from OpenAI

Mutation Operators

Name	Description	Example
AOR	Arithmetic Operator Replacement	a + b → a − b
LVR	Literal Value Replacement	10 → 9
ROR	Relational Operator Replacement	a < b → a >= b
STD	Statement Deletion	remove one line

RQ1: Impact of Mutation Operators

MCT Score for Different Mutation Operators

Model	AOR	LVR	ROR	STD	Avg
GPT-3.5	38.7	20.2	25.7	51.3	34.0
GPT-4	90.3	83.5	81.5	85.2	85.1
Diff	51.6	63.3	55.8	33.9	51.1

- GPT-4 significantly outperforms GPT-3.5 for all mutation operators.
- However, GPT-4 shows the lowest score of 81.5 for ROR, indicating the relative weakness of dealing with relational logic.

RQ2: Impact of Programming Language

MCT Score for Different Programming Language

Model	Python	C++	Go	Java	JS	Rust	Avg
GPT-3.5	29.0	28.8	30.5	27.5	44.8	43.3	34.0
GPT-4	89.3	89.0	81.3	77.8	86.0	87.5	85.1
Diff	60.3	60.2	50.8	50.3	41.2	44.2	51.1

- GPT-4 significantly outperforms GPT-3.5 for all programming languages.
- GPT-4's performance gain is particularly notable in languages with simple syntax and semantics, such as **Python** and **C++**.

RQ4: Impact of Prompting

- Simply using one-shot prompts significantly enhances the performance of GPT-3.5.
- This highlights the effectiveness of providing even minimal contextual guidance to LLMs to improve their accuracy and adaptability in complex tasks.

Language	MO	Zero-Shot	One-Shot	Impr
Python	AOR	42	93	51
	LVR	25	89	64
	ROR	12	70	58
	STD	35	89	54
C++	AOR	23	80	57
	LVR	13	79	66
	ROR	21	85	64
	STD	40	81	41
Go	AOR	32	87	55
	LVR	11	79	68
	ROR	33	85	52
	STD	38	81	43
Java	AOR	34	84	50
	LVR	13	61	48
	ROR	15	63	48
	STD	52	78	26
JavaScript	AOR	45	93	48
	LVR	29	90	61
	ROR	62	94	32
	STD	55	90	35
Rust	AOR	56	100	44
	LVR	33	100	67
	ROR	37	99	62
	STD	35	95	60
Avg	-	32.96	85.21	52.25

Open Questions

Open Questions

What is the impact of using sampled mutants in MCT?

- Initially, we had 38,018 mutants generated but randomly sampled 2,400 mutants in our study.
- What would be the impact of the random sampling? Can we do better?

Open Questions

What is the impact of using sampled mutants in MCT?

- Initially, we had 38,018 mutants generated but randomly sampled 2,400 mutants in our study.
- What would be the impact of the random sampling? Can we do better?

What is the relationship among the various capabilities of LLMs?

- For example, the better the writing code, the better the reading/understanding code?

Open Questions

What is the impact of using sampled mutants in MCT?

- Initially, we had 38,018 mutants generated but randomly sampled 2,400 mutants in our study.
- What would be the impact of the random sampling? Can we do better?

What is the relationship among the various capabilities of LLMs?

- For example, the better the writing code, the better the reading/understanding code?

Can we improve the performance of LLMs using the MCT results?

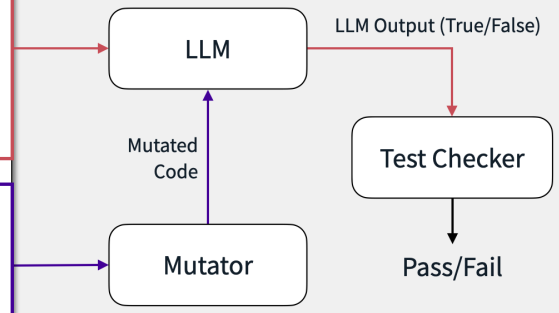
- How can we leverage the MCT results (e.g., the incorrectly identified code-description pairs)?

Overview: Mutation-based Consistency Testing (MCT) for LLMs

```
task_id: HumanEval/155
entry_point: even_odd_count
prompt:
def even_odd_count(num):
    """Given an integer. return a tuple that has
       the number of even and odd digits respectively.

    Example:
       even_odd_count(-12) ==> (1, 1)
       even_odd_count(123) ==> (1, 2)
    """
canonical_solution:
    even, odd = 0, 0
    for ch in str(num):
        if ch in "02468": even += 1
        if ch in "13579": odd += 1
    return even, odd
base_input: [[7], [-78], [3452], [346211], [-345821], ...]
```

- To measure **consistency checking capability**
- Focusing on code **reading**



Case Study Design

RQ1. How do different LLMs fare in terms of MCT for different **mutation operators**?

RQ2. How do different LLMs fare in terms of MCT for different **programming languages**?

RQ3. Can we explain the MCT results of LLMs in terms of **input characteristics**?

RQ4. How do the MCT results of LLMs vary with **zero-shot** and **one-shot promptings**?

HumanEval-X

164 Programming tasks for six languages
(Python, C++, Java, Go, JavaScript (JS), and Rust)

GPT-3.5 & GPT-4

Two most popular LLMs from OpenAI

Mutation Operators

Name	Description	Example
AOR	Arithmetic Operator Replacement	a + b → a − b
LVR	Literal Value Replacement	10 → 9
ROR	Relational Operator Replacement	a < b → a >= b
STD	Statement Deletion	remove one line

RQ1: Impact of Mutation Operators

MCT Score for Different Mutation Operators

Model	AOR	LVR	ROR	STD	Avg
GPT-3.5	38.7	20.2	25.7	51.3	34.0
GPT-4	90.3	83.5	81.5	85.2	85.1
Diff	51.6	63.3	55.8	33.9	51.1

- GPT-4 significantly outperforms GPT-3.5 for all mutation operators.
- However, GPT-4 shows the lowest score of 81.5 for ROR, indicating the relative weakness of dealing with relational logic.

Donghwan Shin

Lecturer (Assistant Professor)

d.shin@sheffield.ac.uk

https://dshin.info





University of Sheffield

Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs

Background: Mutation Analysis

Measure the fault detection capability of a test suite using artificial faults (mutants).

RQ3: Input Characteristic Analysis - Methodology

Build a **decision tree** to infer how the **MCT results** (correct or not) relate to **input characteristics** (features), such as:

- Model (nominal): GPT-3.5 or GPT-4
- Mutation operator (nominal): AOR, ROR, LVR, or STD
- Programming language (nominal): Python, C++, Java, Go, JS, or Rust
- Line (numeric): Total lines of code
- Token (numeric): Total number of tokens in the code
- Mutation position (numeric): Position (line number) of mutation in the code

We used the data collected for RQ1 and RQ2 to build decision trees.

RQ3: Input Characteristic Analysis - Results

