International Journal on Software Tools for Technology Transfer (2026) 28:329–343
https://doi.org/10.1007/s10009-026-00842-4

GENERAL

Special Issue CSV 2025

Understanding code semantics: a benchmark study of LLMs

Cosimo Laneve[1] · Alvise Spanò[2] · Dalila Ressi[2] · Sabina Rossi[2] · Michele Bugliesi[2]

Accepted: 20 February 2026 / Published online: 27 March 2026

Abstract

We present an empirical study on the ability of Large Language Models (LLMs) to understand code by detecting semantically equivalent and inequivalent programs, that is, whether they compute the same result given the same input or not. To probe this, we deliberately perturb the program text by introducing semantics-preserving code transformations, namely copy propagation and constant folding. Using a benchmark of 11 Python functions with both equivalent and non-equivalent variants, we evaluate seven state-of-the-art LLMs (including ChatGPT, Claude, Gemini, and Deep-Seek) under zero-shot prompting, with and without minimal context. Despite strong performance in code generation tasks, the models often fail in this deeper reasoning challenge, misclassifying 41% of equivalent cases without context and 29% with context. Although prompting can improve performance, it does not address the underlying limitations of the models. We argue that improving LLMs themselves, through targeted fine-tuning, contrastive learning on equivalent and nonequivalent implementations, or training on transformation-invariant code, will be necessary for robust semantic understanding. Meanwhile, practitioners can achieve better results by selecting stronger models, carefully engineering prom-pts, or writing code with tools that normalize low-level differences before inference.

Keywords Large language models · Semantic preserving code transformations · Code understanding

1 Introduction

Modern Large Language Models (LLMs) exhibit remarkable capabilities in tasks related to programming, including code generation, comprehension, processing, and analysis [4]. Tools like ChatGPT [23], GitHub Copilot [6], and Amazon Q Developer [2] (previously known as CodeWhisperer) are widespread both in academy and industry to generate, optimize, and fix programs, as well as for tasks such as vulnerability detection and malware analysis.

However, no thorough assessment appears to have been conducted on their robustness in scenarios requiring a nuanced semantics understanding of the code. While LLMs excel at handling syntactic patterns, high-level semantics—such as equivalence of functionally identical programs—is a fundamentally harder problem. In fact, while syntactic correctness and source-level transformations such as variable renaming or introduction of no-op instructions have been explored to evaluate the robustness of LLMs [14, 25, 26], their ability to grasp program equivalence remains underexamined when it comes to non-trivial semantic-preserving transformations and more general semantic equivalence questions.

Semantic equivalence lies at the heart of a wide range of real-world software engineering tasks, including program optimization, refactoring, plagiarism detection, and vulnerability analysis [5]. A system that fails to recognize that two semantically identical programs behave the same may offer incorrect suggestions, overlook bugs, or fail to generalize learned patterns. Despite its importance, this form of reasoning challenges LLMs due to the infinite space of equivalent

✉ C. Laneve
cosimo.laneve@unibo.it

A. Spanò
alvise.spano@unive.it

D. Ressi
dalila.ressi@unive.it

S. Rossi
sabina.rossi@unive.it

M. Bugliesi
michele.bugliesi@unive.it

[1] DISI, University of Bologna, Bologna, Italy

[2] DAIS, Ca' Foscari University of Venice, Venezia Mestre, Italy

programs and the undecidability of equivalence in the general case.

As the theoretical space of semantically equivalent programs is infinite – and determining equivalence is undecidable – we narrow our focus to a specific case of transformations that guarantee identical outputs for identical inputs. In particular, we examine source-to-source transformations inspired by standard compiler-level optimizations that enhance code performance: *copy propagation* and *constant folding* of complex expressions [1].

Although these optimizations are well understood and routinely applied by compilers, they pose a non-trivial challenge to LLMs. Unlike humans or formal tools that rely on symbolic reasoning, LLMs must infer equivalence from learned statistical patterns. This disconnect makes such transformations an ideal testbed for evaluating whether LLMs "understand" code at more than just the surface level.

Our assessment follows the methodology outlined below.

1. First, we formalize copy propagation and constant folding as source-level transformations for a subset of the Python language. We target Python as widely adopted for model training and testing, and restrict to a subset to ease the formalization of the transformations.
2. We collect a benchmark of non-trivial use cases, and code them in the chosen Python subset: the use cases include classic general-purpose algorithms of which the LLMs are well aware, such as Fibonacci or Eratosthenes' Sieve; popular operations on arrays and lists typically found in libraries; and other more domain-specific algorithms such as anti-aliasing and type unification.
3. We apply systematic perturbations to the source samples in the benchmark so that the original programs can be recovered by applying the source-level transformations of interest. By analyzing whether the LLMs can correctly judge the equivalence between the perturbed and original versions, we gain insight into their ability to reason about code and detect deeper relationships than superficial syntactic reorderings.
4. We conduct our assessment on seven mainstream LLMs evaluating their performance based on two classes of prompts. A first, naive prompt simply asks to test the equivalence between the reference samples and their perturbed versions; a second, slightly refined prompt informs the LLMs that the perturbations may be derived by either copy propagation, constant folding, or combinations thereof, without further specifying the specific implementation of the transformations.

Our findings reveal that LLMs are inaccurate in recognizing the perturbed, yet semantically equivalent, variants of programs in 41% of the cases when no context is provided and in about 29% of the cases when given a simple additional context. Among the different models, Anthropic Claude proves to be the best performer, with average failure rates of 19%, while Gemini is the worst, failing in 55% of the cases. As a further experiment, we executed the same tests by adding to the prompt semantic-breaking perturbations. Interestingly, all LLMs performed better, somehow reinforcing the common insight that detecting a mistake is easier than proving correctness. In this case, the overall performance improves, yielding a 24% error rate with drops to only 18% for the contextual prompt.

Collectively, the reported results provide clear evidence that LLMs are still far from robust for evaluating non-trivial transformations, even when confined to specific, well-defined cases. The poor performance observed is likely due to insufficient training for the task, and additional training focused on semantically preserving code transformations is therefore necessary for improvement. However, traditional training methods would be slow and costly, as they would require expert supervision to assess semantic equivalence across a vast code base, even for specific transformations like constant folding. To lower both the learning curve and associated costs, we advocate for integrating LLMs with automatic tools that implement code transformations, such as those proposed in this paper. Such integration would serve two complementary purposes. First, it would make the tools available for self-supervised training starting from the existing code base (simply, the tools would be employed to produce semantic equivalent transformations of the existing samples in the code base). Secondly, and more interestingly, they could be employed as general-purpose code pre-processors to eliminate "noisy" bits, thus enhancing LLMs in their code understanding task.

Paper structure The rest of the paper is structured as follows. Section 2 reports and discusses related works. Section 3 presents in detail our methodology, namely the two code transformations (copy propagation and constant folding), the definition of the dataset and the prompts. Section 4 reports our results and Sect. 5 discusses them, analyzing possible solutions to improve reliability. We conclude in Sect. 6 where we also present future work.

2 Related work

The integration of formal methods tools with Artificial Intelligence techniques and Machine Learning (ML) models is an emerging practice in various domains [35], including data augmentation [24], dataset labeling [30], network compression [27, 29], and security [28, 30]: formal methods provide a structured approach to defining and validating semantic-preserving transformations, while Artificial Intelligence techniques help enhance the scalability of the verification processes.

LLM models are increasingly being evaluated in their performance with code generation tasks. In particular, some studies assess code generation and manipulation of Github Copilot against selected fundamental algorithms such as sorting and data structure implementation [7, 36]. Other works aim to reduce the number of hallucinations (when LLMs generate confident-sounding, fluent, yet factually incorrect, nonsensical, or fabricated information not supported by their training data or reality) related to coding tasks [21, 33].

Our present focus is more directly related to the recent stream of research work that investigates the performance of LLMS in code analysis and bug detection. In particular, the results by [9, 11] align with our own findings, confirming that even trivial modifications, such as variable renaming, can lead to performance degradation. On a different, but still related account, [3] analyzes ML-based attacks on fully homomorphic encryption via side channels, using code transformation techniques to show that unoptimized code hinders such attacks.

However, to our knowledge, no prior work has systematically analyzed the impact of fine-grained optimizations such as the ones of our present interest. We should also mention that the analysis itself is challenging as the evaluation metrics depend on several factors, including both the varying prompting functionalities offered by the models and the often nuanced format of the responses [22].

A key distinction in chatbot evaluation is between *user prompts* and *system prompts* [31]. User prompts do not require the API and include basic prompting, zero or few-shot prompting, chain-of-thought prompting, and role-based or instruction-tuned prompting. System prompts, in turn, support customization of behavior, max token limits, temperature setting, and further advanced features like function calling and logit bias, and streaming responses [31].

Our assessment is based on user prompts rather than system prompts for two main reasons. Chatbot evaluation distinguishes between user prompts and system prompts [31]. User prompts, which do not require API access, include various prompting techniques, while system prompts enable predefined behaviors and advanced settings. We focus on user prompts for two reasons. Firstly, free-tier chatbots, like ChatGPT, are widely used and reflect real-world interactions more accurately than open-source models like LLaMA [10], Gemma [34], and Mixtral [17], which are primarily used in research. Secondly, system prompts involve hidden parameters (e.g., temperature), making results less consistent. Additionally, we employ a zero-shot approach [18], assessing chatbots without providing prior context or examples.

3 Methodology

To evaluate the ability of ML models to understand and manipulate source code, we selected Python as the target programming language for our study. Its simplicity and widespread adoption among developers make it a common focus in model training, enabling effective analysis and reasoning. In the following subsection, we describe our techniques for code perturbation, dataset construction, and the experimental setup.

3.1 Semantic-preserving code optimizations

As discussed in the introduction, we perturb Python code in a way that requires nuanced semantic understanding. To ensure the correctness of these transformations, we leverage two well-known compiler techniques commonly used in intermediate code optimization: *copy propagation* of variables and *constant folding* of expressions. To apply these techniques at the source code level, we

(1) use *annotations* to record information about variables (whether they are either constant values or copies);
(2) bind annotations to every control point of the code (every instruction) by means of a type inference system and a fixpoint analysis;
(3) simplify the code according to the information stored in the annotations.

The most technical aspect of this procedure is point (2), which operates as follows: (*i*) we define a set of rules that modify annotations based on their corresponding instructions; (*ii*) starting with a program where each instruction has an empty annotation, we iteratively infer annotations; (*iii*) the inference process continues until no further modifications occur (*i.e.*, a fixpoint is reached); (*iv*) once the fixpoint is reached, we proceed with the transformation step that simplifies the original program. For a formal definition of point (2), we provide typing rules for a subset of Python statements – sufficient to cover the codes in Sect. 4. The correctness of the overall process is beyond the scope of this paper; however, it can be proven using a similar approach to that presented in [1] for corresponding optimization techniques applied to intermediate code.

The two transformations are defined below. We use the following notations: given a Python program $\mathtt{P}$, let $Var(\mathtt{P})$ be the set of variables of $\mathtt{P}$ ($Var(\cdot)$ also applies to statements S, with the same meaning); if S is either a statement or an expression, we write $S\{^{y}/_{x}\}$ the substitution of every occurrence of x in S with y (we assume that x and y are not redefined in S, i.e., they are *free*).

3.1.1 Copy propagation

The annotations of the copy propagation analysis are sets $\mathbb{P}$ of pairs (x, y), with $x, y \in Var(\mathtt{P})$, that are closed by symmetry and transitivity. We always abbreviate $\{(x, y), (y, x)\}$ with $x \sim y$ whose meaning is "*x and y store the same value, i.e., they are copies*". Let

Table 1 Rewriting rules for Copy Propagation: they define how to derive judgments of the form $\mathbb{P} \vdash S \blacktriangleright \mathbb{P}'$, which are interpreted as follows: if the statement S is evaluated in a memory where the variables in $\mathbb{P}$ are copies, then its evaluation terminates in a memory where the variables in $\mathbb{P}'$ are also copies

$$\text{[Asgn-CP]}\ \frac{E \text{ is not a variable}}{\mathbb{P} \vdash x = E \blacktriangleright \mathbb{P} \setminus x} \qquad \text{[Id-CP]}\ \frac{x \neq y}{\mathbb{P} \vdash x = y \blacktriangleright (\mathbb{P} \setminus x \cup \{(x,y),(y,x)\})^{\mathtt{st}}} \qquad \text{[Ids-CP]}\ \mathbb{P} \vdash x = x \blacktriangleright \mathbb{P}$$

$$\text{[If-CP]}\ \frac{\mathbb{P} \vdash S \blacktriangleright \mathbb{P}' \qquad \mathbb{P} \vdash S' \blacktriangleright \mathbb{P}''}{\mathbb{P} \vdash \mathtt{if}\ (E) :\ S\ \mathtt{else}:\ S' \blacktriangleright \mathbb{P}' \cap \mathbb{P}''} \qquad \text{[While-CP]}\ \frac{\mathbb{P} \cap \mathbb{P}' \vdash S \blacktriangleright \mathbb{P}'}{\mathbb{P} \vdash \mathtt{while}\ (E) :\ S \blacktriangleright \mathbb{P} \cap \mathbb{P}'}$$

$$\text{[For-CP]}\ \frac{\mathbb{P} \cap \mathbb{P}' \setminus i \vdash S \blacktriangleright \mathbb{P}'}{\mathbb{P} \vdash \mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(E) :\ S \blacktriangleright \mathbb{P} \cap \mathbb{P}' \setminus i} \qquad \text{[Seq-CP]}\ \frac{\mathbb{P} \vdash S \blacktriangleright \mathbb{P}' \qquad \mathbb{P}' \vdash S' \blacktriangleright \mathbb{P}''}{\mathbb{P} \vdash S\ S' \blacktriangleright \mathbb{P}''}$$

$$\text{[Def-CP]}\ \frac{\emptyset \vdash S \blacktriangleright \mathbb{P}'}{\mathbb{P} \vdash \mathtt{def}\ f(x_1,\ldots,x_n) :\ S \blacktriangleright \mathbb{P}} \qquad \text{[Call-CP]}\ \mathbb{P} \vdash f(e_1,\ldots,e_n) \blacktriangleright \mathbb{P}$$

- $\mathbb{P} \setminus x \stackrel{\text{def}}{=} \{(y,z) \in \mathbb{P} \mid y \neq x \text{ and } z \neq x\}$;
- $\mathbb{P} \cap \mathbb{P}' \stackrel{\text{def}}{=} \{(x,y) \mid (x,y) \in \mathbb{P} \text{ and } (x,y) \in \mathbb{P}'\}$;
- $(A)^{\mathtt{st}}$ is the closure of a set A by symmetry and transitivity.

The inference system presented in Table 1 defines how to derive judgments of the form $\mathbb{P} \vdash S \blacktriangleright \mathbb{P}'$, which are interpreted as follows: if the statement S is evaluated in a memory where the variables in $\mathbb{P}$ are copies, then its evaluation terminates in a memory where the variables in $\mathbb{P}'$ are also copies.

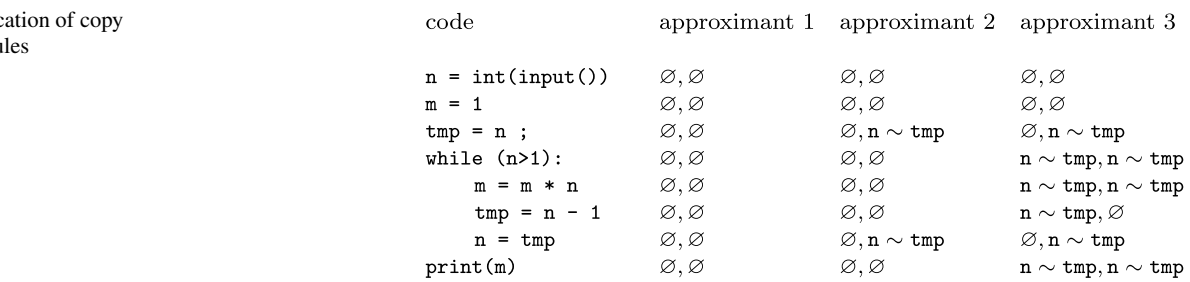
We now briefly describe the most relevant rules used to derive the set of copies of a Python program P, as outlined in Table 1.

Rule [Asgn-CP] defines the annotation of an assignment to a variable x. In this case (E is not a variable), any copy of x is broken and must be removed from the set $\mathbb{P}$, hence the conclusion $\mathbb{P} \setminus x$. Rule [Id-CP] destroys previous copies of x and creates a copy with y and with all the variables that are copies of y in $\mathbb{P}$. Rule [While-CP] is the critical one. To determine the set $\mathbb{P}''$ such that $\mathbb{P} \vdash \mathtt{while}\ (E) :\ S \blacktriangleright \mathbb{P}''$ we need an invariant: a set $\mathbb{P}'$ such that $\mathbb{P}' \vdash S \blacktriangleright \mathbb{P}'$. However, a generic set $\mathbb{P}'$ is not correct because we cannot have more copies than those when the control reaches the first time the iteration. Hence the invariant becomes $\mathbb{P} \cap \mathbb{P}' \vdash S \blacktriangleright \mathbb{P}'$ and the conclusion of [While-CP] takes into account this invariant and the fact that the iteration may be not executed at all. Rule [For-CP] is similar to [While-CP]. Rules [Def-CP] and [Call-CP] deal with function definitions and invocations; they assume that no access to global variables is possible. With this constraint, the judgments are pretty easy.

The main difficulty of the system in Table 1 is determining the invariants in correspondence of iterations. There is a standard way to solve this issue: starting with empty annotations, running the type systems several times until the annotations do not change anymore. This algorithm always terminates (given a program, the corresponding sets $\mathbb{P}$ are a finite lattice and the iterations always return greater annotations). As an example, we apply the algorithm to a simple code printing the factorial of a value taken in input, shown in Fig. 1. Hereafter, we abbreviate $\mathbb{P} \vdash S \blacktriangleright \mathbb{P}'$ with $S\ \mathbb{P},\mathbb{P}'$). In this case, the algorithm terminates after three applications of the type system in Table 1 corresponding to the three approximants in the Figure.

We now illustrate in detail the code transformations obtained by applying the rules in Table 1. In these transformations, we adopt a slight abuse of notation by using semicolons to separate an assignment from its continuation. Specifically, Python code of the form $x = E\ \ S$ is written as $x = E\ ;\ S$. The code transformations induced by the copy propagation analysis, denoted as $\stackrel{\text{CP}}{\Longrightarrow}$, are as follows:

1. if $\mathbb{P} \vdash x = y \blacktriangleright \mathbb{P}'$ with $x \sim y \subseteq \mathbb{P}$ then
$$x = y \quad \stackrel{\text{CP}}{\Longrightarrow} \quad \varepsilon$$
(in this case $\mathbb{P} = \mathbb{P}'$: we are erasing $x = y$);
2. if $\mathbb{P} \vdash x = y \blacktriangleright \mathbb{P}'$ and $\mathbb{P}' \vdash S \blacktriangleright \mathbb{P}''$, such that either ($i$) every annotation in the proof of $\mathbb{P}' \vdash S \blacktriangleright \mathbb{P}''$ contains $x \sim y$ or (ii) $x \notin Var(S)$, then
$$x = y\ ;\ S \quad \stackrel{\text{CP}}{\Longrightarrow} \quad S\{^{y}/_{x}\}\ ;\ x = y$$
3. if $\mathbb{P} \vdash y = e\ ;\ S\ ;\ x = y \blacktriangleright \mathbb{P}'$ and $x \sim y \subseteq \mathbb{P}$ and $x \notin Var(S)$ then
$$y = e\ ;\ S\ ;\ x = y \quad \stackrel{\text{CP}}{\Longrightarrow} \quad x = e\ ;\ S\{^{x}/_{y}\}\ ;\ y = x \qquad (1)$$
(in this case $x \sim y \subseteq \mathbb{P}'$)
4. if $\mathbb{P} \vdash \mathtt{while}\ (e) :\ (S\ ;\ x = y) \blacktriangleright \mathbb{P}'$ and $(x,y) \in \mathbb{P}$ and $x \notin Var(\mathtt{S})$ then
$$\mathtt{while}\ (e) : (S\ ;\ x = y) \quad \stackrel{\text{CP}}{\Longrightarrow} \quad (\mathtt{while}\ (e\{^{y}/_{x}\}) : S)\ ;\ x = y \qquad (2)$$
5. if $\mathbb{P} \vdash \mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(e) :\ (S\ ;\ x = y) \blacktriangleright \mathbb{P}'$ and $x \sim y \subseteq \mathbb{P}$ and $x \notin Var(S) \setminus i$ then
$$\mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(e) : (S\ ;\ x = y) \quad \stackrel{\text{CP}}{\Longrightarrow} \quad (\mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(e\{^{y}/_{x}\}) : S)\ ;\ x = y \qquad (3)$$



code	approximant 1	approximant 2	approximant 3
`n = int(input())`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\varnothing, \varnothing$
`m = 1`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\varnothing, \varnothing$
`tmp = n ;`	$\varnothing, \varnothing$	$\varnothing, \mathtt{n} \sim \mathtt{tmp}$	$\varnothing, \mathtt{n} \sim \mathtt{tmp}$
`while (n>1):`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\mathtt{n} \sim \mathtt{tmp}, \mathtt{n} \sim \mathtt{tmp}$
`  m = m * n`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\mathtt{n} \sim \mathtt{tmp}, \mathtt{n} \sim \mathtt{tmp}$
`  tmp = n - 1`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\mathtt{n} \sim \mathtt{tmp}, \varnothing$
`  n = tmp`	$\varnothing, \varnothing$	$\varnothing, \mathtt{n} \sim \mathtt{tmp}$	$\varnothing, \mathtt{n} \sim \mathtt{tmp}$
`print(m)`	$\varnothing, \varnothing$	$\varnothing, \varnothing$	$\mathtt{n} \sim \mathtt{tmp}, \mathtt{n} \sim \mathtt{tmp}$

Fig. 1 Application of copy propagation rules

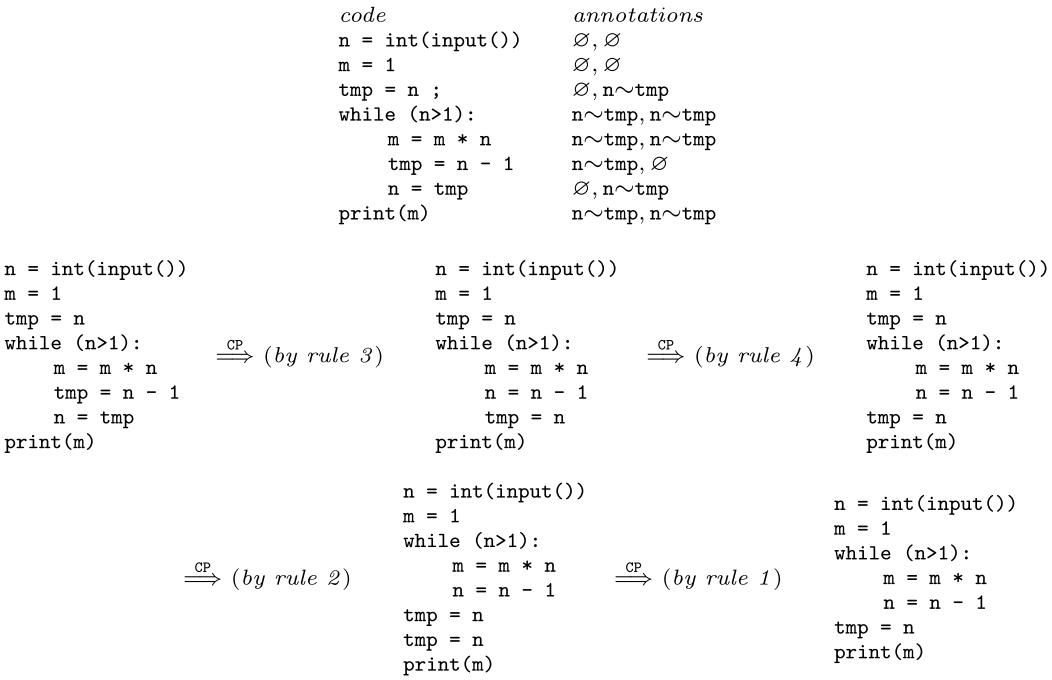


Fig. 2 Annotations and transformations of copy propagation

For example, in Fig. 2, we show the code transformation of our simple iterative program computing the factorial. At each step, we indicate the transformation rule that has been applied. In particular, in the first step, we apply rule 3 when S is empty. It is worth to notice that since every transformation step modifies the program, the corresponding annotations must also be updated. However, rerunning the inference system at every step is not necessary; instead, annotation updates can be incorporated directly within the transformation steps. Finally, we notice that it is possible to use the standard garbage collection rule:

If $\mathtt{P} = S\ \mathtt{;}\ x = e\ \mathtt{;}\ S'$ and $x \notin Var(S')$ then remove $x = e$.

For example, in the last code of Fig. 2, we can delete the statement `tmp = n`.

3.1.2 Constant folding

The annotations for the constant folding are *abstract memories* $\mathbb{C}$, which are sets of pairs (x, μ) where $x \in Var(\mathtt{P})$ (for some Python program P) and μ is either a constant value or $\top$, where $\top$ represents a non-constant value, or ? that represents an error. We use the following operations:

- $\mathbb{C} \setminus x \stackrel{\text{def}}{=} \{(y, \mu) \in \mathbb{C} \mid y \neq x\}$;
- $\mu \sqcup \mu' \stackrel{\text{def}}{=} \begin{cases} \mu & \text{if } \mu = \mu' \\ \top & \text{if } \mu \neq \mu' \text{ and } \mu, \mu' \neq ? \\ ? & \text{if } \mu = ? \text{ or } \mu' = ? \end{cases}$
- $\mathbb{C} \sqcup \mathbb{C}' \stackrel{\text{def}}{=} \begin{cases} \{(x, \mu \sqcup \mu')\} \cup (\mathbb{C} \setminus x \sqcup \mathbb{C}' \setminus x) \\ \qquad \text{if } (x, \mu) \in \mathbb{C} \text{ and } (x, \mu') \in \mathbb{C}' \\ \{(x, \mu)\} \cup (\mathbb{C} \setminus x \sqcup \mathbb{C}') \\ \qquad \text{if } (x, \mu) \in \mathbb{C} \text{ and } \mathbb{C}' \setminus x = \mathbb{C}' \\ \{(x, \mu')\} \cup (\mathbb{C} \sqcup \mathbb{C}' \setminus x) \\ \qquad \text{if } (x, \mu') \in \mathbb{C}' \text{ and } \mathbb{C} \setminus x = \mathbb{C} \end{cases}$
- $[\![E]\!]_{\mathbb{C}}$ that evaluates E in the abstract memory $\mathbb{C}$; if some variable in E has value ? in $\mathbb{C}$ or has no value then the result is ? (error); otherwise if some variable is $\top$ (and no variable is ? or has no value in $\mathbb{C}$) then the overall result is $\top$; otherwise the result is the value of E when all the

Table 2 Rewriting rules for Constant Folding: they define how to derive judgments of the form $\mathbb{C} \vdash S \blacktriangleright \mathbb{C}'$ whose meaning is: if S is evaluated in an abstract memory where variables have the values stored in $\mathbb{C}$ then its evaluation terminates with a memory whose variables have the values stored in $\mathbb{C}'$

$$\text{[Asgn-CF]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mu}{\mathbb{C} \vdash x = E \blacktriangleright \mathbb{C} \setminus x \cup \{(x, \mu)\}} \qquad \text{[If-CF]}\ \frac{[\![E]\!]_{\mathbb{C}} = \top \quad \mathbb{C} \vdash S \blacktriangleright \mathbb{C}' \quad \mathbb{C} \vdash S' \blacktriangleright \mathbb{C}''}{\mathbb{C} \vdash \mathtt{if}\ (E) :\ S\ \mathtt{else} :\ S' \blacktriangleright \mathbb{C}' \sqcup \mathbb{C}''}$$

$$\text{[If-CF-t]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mathit{true} \quad \mathbb{C} \vdash S \blacktriangleright \mathbb{C}'}{\mathbb{C} \vdash \mathtt{if}\ (E) :\ S\ \mathtt{else} :\ S' \blacktriangleright \mathbb{C}'} \qquad \text{[If-CF-f]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mathit{false} \quad \mathbb{C} \vdash S' \blacktriangleright \mathbb{C}''}{\mathbb{C} \vdash \mathtt{if}\ (E) :\ S\ \mathtt{else} :\ S' \blacktriangleright \mathbb{C}''}$$

$$\text{[While-CF]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mu \quad \mu \neq \mathit{false} \quad \mathbb{C} \sqcup \mathbb{C}' \vdash S \blacktriangleright \mathbb{C}'}{\mathbb{C} \vdash \mathtt{while}\ (E) :\ S \blacktriangleright \mathbb{C} \sqcup \mathbb{C}'} \qquad \text{[While-CF-f]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mathit{false} \quad \mathbb{C} \sqcup \mathbb{C}' \vdash S \blacktriangleright \mathbb{C}'}{\mathbb{C} \vdash \mathtt{while}\ (E) :\ S \blacktriangleright \mathbb{C}}$$

$$\text{[For-CF]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mu \quad (\mu > 0 \text{ or } \mu = \top) \quad \mathbb{C} \sqcup \mathbb{C}' \sqcup \{(i, \top)\} \vdash S \blacktriangleright \mathbb{C}'}{\mathbb{C} \vdash \mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(E) :\ S \blacktriangleright \mathbb{C} \sqcup \mathbb{C}' \sqcup \{(i, \top)\}} \qquad \text{[For-CF-f]}\ \frac{[\![E]\!]_{\mathbb{C}} = \mathtt{k} \quad \mathtt{k} \leq 0}{\mathbb{C} \vdash \mathtt{for}\ i\ \mathtt{in}\ \mathtt{range}(E) :\ S \blacktriangleright \mathbb{C}}$$

$$\text{[Seq-CF]}\ \frac{\mathbb{C} \vdash S \blacktriangleright \mathbb{C}' \quad \mathbb{C}' \vdash S' \blacktriangleright \mathbb{C}''}{\mathbb{C} \vdash S\ S' \blacktriangleright \mathbb{C}''} \qquad \text{[Def-CF]}\ \frac{\top \vdash S \blacktriangleright \mathbb{C}'}{\mathbb{C} \vdash \mathtt{def}\ f(x_1, \ldots, x_n) :\ S \blacktriangleright \mathbb{C}} \qquad \text{[Call-CF]}\ \mathbb{C} \vdash f(e_1, \ldots, e_n) \blacktriangleright \mathbb{C}$$

code	approximant 1	approximant 2	approximant 3
`n = int(input())`	$\emptyset,\ \emptyset$	$\emptyset,\ \{(\mathtt{n}, \top)\}$	$\emptyset,\ \{(\mathtt{n}, \top)\}$
`tmp = 1`	$\emptyset,\ \emptyset$	$\{(\mathtt{n}, \top)\},\ \{(\mathtt{n}, \top), (\mathtt{tmp}, 1)\}$	$\{(\mathtt{n}, \top)\},\ \{(\mathtt{n}, \top), (\mathtt{tmp}, 1)\}$
`m = 2*tmp - 1`	$\emptyset,\ \emptyset$	$\{(\mathtt{n}, \top), (\mathtt{tmp}, 1)\},\ \mathbb{C}_1$	$\{(\mathtt{n}, \top), (\mathtt{tmp}, 1)\},\ \mathbb{C}_1$
`while (n > tmp):`	$\emptyset,\ \emptyset$	$\mathbb{C}_1,\ \mathbb{C}_1$	$\mathbb{C}_4,\ \mathbb{C}_4$
`  tmp = tmp + 1`	$\emptyset,\ \emptyset$	$\mathbb{C}_1,\ \mathbb{C}_2$	$\mathbb{C}_4,\ \mathbb{C}_3$
`  m = m * n`	$\emptyset,\ \emptyset$	$\mathbb{C}_2,\ \mathbb{C}_3$	$\mathbb{C}_3,\ \mathbb{C}_3$
`  n = n - tmp + 1`	$\emptyset,\ \emptyset$	$\mathbb{C}_3,\ \mathbb{C}_3$	$\mathbb{C}_3,\ \mathbb{C}_3$
`  tmp = tmp - 1`	$\emptyset,\ \emptyset$	$\mathbb{C}_3,\ \mathbb{C}_4$	$\mathbb{C}_3,\ \mathbb{C}_4$
`print(m)`	$\emptyset,\ \emptyset$	$\mathbb{C}_4,\ \mathbb{C}_4$	$\mathbb{C}_4,\ \mathbb{C}_4$

Fig. 3 Application of constant folding rules ($\mathbb{C}_1 = \{(\mathtt{n}, \top), (\mathtt{tmp}, 1), (\mathtt{m}, 1)\}$, $\mathbb{C}_2 = \{(\mathtt{n}, \top), (\mathtt{tmp}, 2), (\mathtt{m}, 1)\}$, $\mathbb{C}_3 = \{(\mathtt{n}, \top), (\mathtt{tmp}, 2), (\mathtt{m}, \top)\}$, $\mathbb{C}_4 = \{(\mathtt{n}, \top), (\mathtt{tmp}, 1), (\mathtt{m}, \top)\}$)

variables have been replaced by the corresponding values in $\mathbb{C}$ (errors may occur at this stage as well, *e.g.* the division by 0 returns ?).

The *judgments* for the constant folding are $\mathbb{C} \vdash S \blacktriangleright \mathbb{C}'$, meaning that, if S is typed with an initial abstract memory $\mathbb{C}$, then its evaluation terminates with a memory whose variables have the values stored in $\mathbb{C}'$. The rules for deriving copies of a Python program P are written in Table 2. We comment on rules [Asgn-CF] and [If-CF], as the explanations for the remaining rules closely follow those given for copy propagation.

Rule [Asgn-CF] updates the abstract memory $\mathbb{C}$ by binding x with the value $[\![E]\!]_{\mathbb{C}}$. Rule [If-CF] types conditionals when the value of the guard is not determined. In this case the abstract memory after the conditional must include the results $\mathbb{C}'$ and $\mathbb{C}''$ of the two branches (at static time we are not aware of the branches that will be executed). Hence the abstract memory $\mathbb{C}' \sqcup \mathbb{C}''$ is the conclusion.

Like the copy propagation analysis, the constant folding requires an iterative algorithm to determine the annotations. The algorithm starts with empty abstract memories. As an example, we apply the algorithm to a simple code printing the factorial of a value taken in input, shown in Fig. 3. Also in this case, the algorithm terminates after three applications of the type system in Table 2.

The inference system reported in Table 2) allows us to derive judgments of the form $\mathbb{C} \vdash S \blacktriangleright \mathbb{C}'$ whose meaning is: *if S is evaluated in an abstract memory where variables have the values stored in $\mathbb{C}$ then its evaluation terminates with a memory whose variables have the values stored in $\mathbb{C}'$*. The topmost code in Fig. 4 reports the annotations we derive, where we write $S\ \mathbb{C}, \mathbb{C}'$ instead of $\mathbb{C} \vdash S \blacktriangleright \mathbb{C}'$.

The pair $(x, \mathtt{k})$, where $\mathtt{k}$ is a constant, means that the variable x has value $\mathtt{k}$ (in the memory); $(x, \top)$ means that the variable x is undetermined (in the memory) – we are not able to determine its value at static time – or ? that represents an error.

The transformation steps for the constant folding, denoted as $\overset{\text{CF}}{\Longrightarrow}$, use an auxiliary function $[\![E]\!]_{\mathbb{C}}$ that, given an expression E, evaluates E in the abstract memory $\mathbb{C}$; if some variable in E has either no value or it is ? in $\mathbb{C}$ then the result is ? (error), otherwise (some variable is $\top$ in $\mathbb{C}$ and no variable is ?) the result is $\top$. We use *expression contexts*, written $\mathcal{E}[\ ]$, which are expressions with one hole $[\ ]$ in correspondence of a sub-expression. For example $x + [\ ]$ is an

expression context. The steps are defined by the following four rules:

1. if S is either $x = \mathcal{E}[E]$ or $\texttt{if}\,(\mathcal{E}[E]) : S'\ \texttt{else} : S''$ or $\texttt{while}\,(\mathcal{E}[E]) : S'$ or $\texttt{for}\ i\ \texttt{in}\ \texttt{range}(\mathcal{E}[E]) : S'$ or $f(\cdots, \mathcal{E}[E], \ldots)$ and $\mathbb{C} \vdash S \blacktriangleright \mathbb{C}'$ with $[\![E]\!]_{\mathbb{C}} = \mathtt{k}$ and $\mathtt{k} \neq \top$ then S may be rewritten as follows:

$$
\begin{aligned}
x = \mathcal{E}[E] &\overset{\text{CF}}{\Longrightarrow} x = \mathcal{E}[\mathtt{k}] \\
\texttt{if}\,(\mathcal{E}[E]) : S'\ \texttt{else} : S'' &\overset{\text{CF}}{\Longrightarrow} \texttt{if}\,(\mathcal{E}[\mathtt{k}]) : S'\ \texttt{else} : S'' \\
\texttt{while}\,(\mathcal{E}[E]) : S' &\overset{\text{CF}}{\Longrightarrow} \texttt{while}\,(\mathcal{E}[\mathtt{k}]) : S' \\
\texttt{for}\ i\ \texttt{in}\ \texttt{range}(\mathcal{E}[E]) : S' &\overset{\text{CF}}{\Longrightarrow} \texttt{for}\ i\ \texttt{in}\ \texttt{range}(\mathcal{E}[\mathtt{k}]) : S' \\
f(\cdots, \mathcal{E}[E], \ldots) &\overset{\text{CF}}{\Longrightarrow} f(\cdots, \mathcal{E}[\mathtt{k}], \ldots)
\end{aligned}
$$

2. if $\mathbb{C} \vdash x = E \blacktriangleright \mathbb{C}'$ and $[\![E]\!]_{\mathbb{C}} = \mathtt{k}$ $(\mathtt{k} \neq \top)$ and $(x, \mathtt{k}) \in \mathbb{C}$ then

$$x = E \overset{\text{CF}}{\Longrightarrow} \varepsilon$$

(remove the assignment);

3. if $\mathbb{C} \vdash x = E\ ;\ S \blacktriangleright \mathbb{C}'$ and $x \notin Var(S)$ then

$$x = E\ ;\ S \overset{\text{CF}}{\Longrightarrow} S\ ;\ x = E$$

if $\mathbb{C} \vdash x = E\ ;\ x = E' \blacktriangleright \mathbb{C}'$ and $x \notin Var(E')$ then

$$x = E\ ;\ x = E' \overset{\text{CF}}{\Longrightarrow} x = E'$$

4. if either $\mathbb{C} \vdash \texttt{if}\,(E) : S\ \texttt{else} : S' \blacktriangleright \mathbb{C}'$ or $\mathbb{C} \vdash \texttt{while}\,(E) : S \blacktriangleright \mathbb{C}'$ or $\mathbb{C} \vdash \texttt{for}\ i\ \texttt{in}\ \texttt{range}(E) : S \blacktriangleright \mathbb{C}'$ then

$$
\begin{aligned}
\texttt{if}\,(E) : S\ \texttt{else} : S' &\overset{\text{CF}}{\Longrightarrow} S \quad (\text{when } [\![E]\!]_{\mathbb{C}} = \mathit{true}) \\
\texttt{if}\,(E) : S\ \texttt{else} : S' &\overset{\text{CF}}{\Longrightarrow} S' \quad (\text{when } [\![E]\!]_{\mathbb{C}} = \mathit{false}) \\
\texttt{while}\,(E) : S &\overset{\text{CF}}{\Longrightarrow} \varepsilon \quad (\text{when } [\![E]\!]_{\mathbb{C}} = \mathit{false}) \\
\texttt{for}\ i\ \texttt{in}\ \texttt{range}(E) : S &\overset{\text{CF}}{\Longrightarrow} \varepsilon \quad (\text{when } [\![E]\!]_{\mathbb{C}} = \mathtt{k} \text{ and } \mathtt{k} \leq 0)
\end{aligned}
$$

In Fig. 4, we apply the constant folding transformations to our simple iterative program computing the factorial.

In particular, in the first step, we apply rule 1, thus computing all the constant subexpressions. As for copy propagation, it is possible to collect garbage variables not used in the continuations. Therefore, the assignment `tmp = 1` in the last code may be removed.

3.2 Dataset construction

The dataset comprises multiple semantically equivalent and non-equivalent variants of small Python programs, designed to assess the model's ability to distinguish correct from incorrect implementations. We crafted 11 distinct programs in total, each available in 8 variants. These programs consist of standalone functions implementing various algorithms, differing in length, complexity, and notoriety. Some are general-purpose algorithms, such as Bubble Sort and the Sieve of Eratosthenes, while others, like type unification and 3D point rotation, are more domain-specific. The dataset includes both algorithms that manipulate data structures, such as lists and arrays, and those performing mathematical computations, ensuring a diverse set of challenges. Table 3 provides an overview of all included algorithms.

Table 3 Our dataset of Python programs. Green denotes general-purpose classic algorithms; yellow marks notorious functions that are typically found in libraries; red refers to algorithms that belong to specific domains, such as computer graphics or compiler construction

Arithmetics	List Manipulation	Other
Fibonacci	Remove Duplicates	Anti Aliasing
Primality Test	Find Occurrence	Unification (MGU)
Rotate 3D Point	Count Occurrences	
Fast Fourier Transform	Sieve of Eratosthenes	
	Bubblesort	

Each algorithm in the dataset follows a structured design, with implementations grouped into two classes:

- **Correct Implementations (Semantically Equivalent):**
  - One standard implementation serving as a reference.
  - Three perturbed versions that preserve the original semantics while applying simple transformations - copy propagation, constant folding, and a combination of both. These transformations are reversible using standard optimization rules (see Sect. 3.1).
- **Incorrect Implementations (Semantically Non-equivalent):**
  - One version introducing a semantic-breaking bug in the reference implementation.
  - Three additional versions derived from each of the perturbed correct implementations by introducing a bug. These errors involve small but meaningful changes, such as index modifications or incorrect assignments, making them semantically incorrect.

To further challenge model recognition, especially for domain-specific algorithms, we introduced a degree of obfuscation. Function names were anonymized, and variable identifiers were replaced with short, uninformative labels to prevent reliance on superficial patterns.

The dataset is publicly available on GitHub [32]. Each source file contains one algorithm in the 8 variants, plus some extra code that puts the actual semantic equivalence to the test by performing unit tests for a reasonable range of inputs and comparing the outputs.

3.3 Experimental setup

Our experiments involved collecting and analyzing responses from several mainstream chatbots. We formulated our case

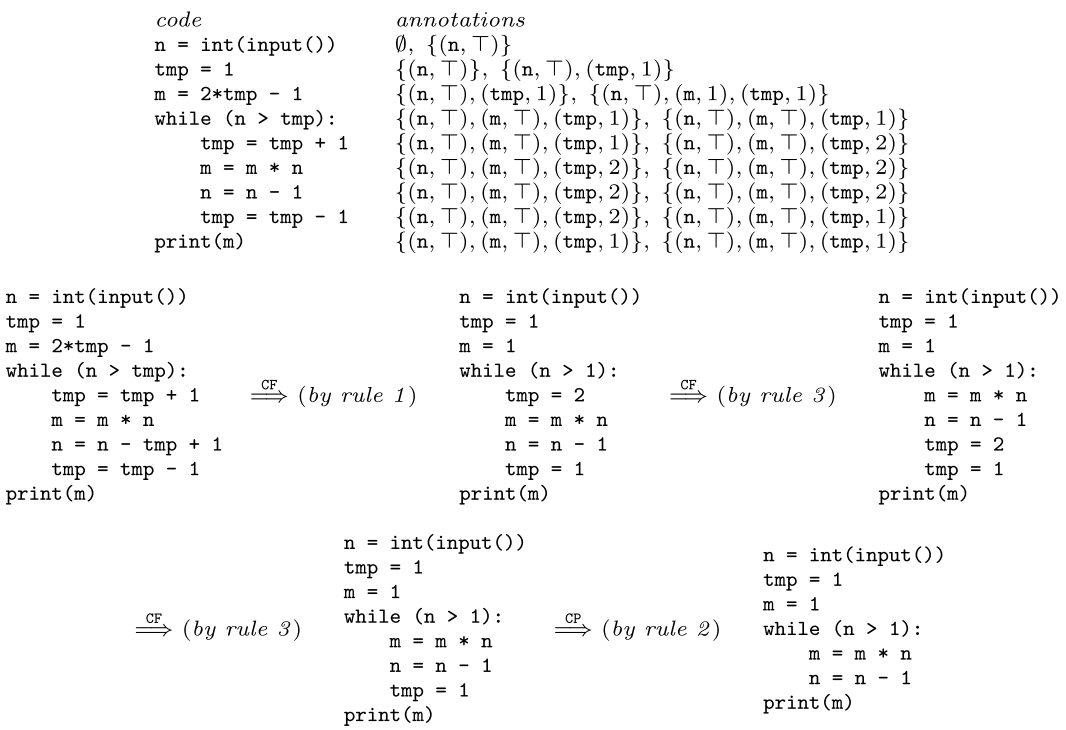


Fig. 4 Annotations and transformations of constant folding

study as a multi-instance binary classification problem, where both the reference implementation and its perturbed variants were presented within the same prompt. For each algorithm in the dataset, we designed four zero-shot prompts as follows:

	Contextless Preamble	Contextual Preamble
Single-Class	Prompt 1	Prompt 2
Multi-Class	Prompt 3	Prompt 4

Prompt 1 (single-class, contextless): a contextless brief question, followed by the four correct versions of the algorithm (one reference and three perturbed);

Prompt 2 (single-class, contextual): a contextualized preamble, followed by the same four correct versions;

Prompt 3 (multi-class, contextless): Prompt 1 plus the four incorrect implementations, resulting in a total of eight code snippets that mix correct and incorrect versions;

Prompt 4 (multi-class, contextual): similar to Prompt 3, but including the contextual preamble from Prompt 2.

Prompt 1 is intended to ask: *"Can the chatbot recognize correct implementations based solely on structure, without contextual guidance?"* It presents a brief, contextless question alongside four correct implementations—the reference and three perturbed versions—testing whether the chatbot can assess correctness purely from code patterns. Prompt 2 adds background information to evaluate whether context improves classification accuracy. Prompt 3 shifts focus to: *"Can the chatbot distinguish between correct and incorrect implementations with no contextual support?"* Unlike Prompt 1, it includes both valid and flawed versions, requiring the model to rely solely on its understanding of correctness. Prompt 4 combines this challenge with additional context, assessing whether more information aids or complicates classification.

Contextless Preamble

```
Are the following functions semantically equivalent to
the first one?
```

Contextual Preamble

```
You are a chatbot for comparing the semantics of small
Python programs. I will provide you with multiple
implementations of the same Python function. The first
function is the reference version. The other functions
are perturbed with copy propagation, constant folding or
a combination of the two. Tell me whether the functions
are semantically equivalent to the reference version or
not.
```

Fig. 5 Difference between the preamble of the contextless and contextual prompts

Each response to the prompts was carefully reviewed by two of the authors and framed as a binary multi-instance classification problem. Specifically, each algorithm within a prompt was compared to the first function provided, which

served as the reference to assess semantic equivalence. To evaluate the explanations produced by the chatbots, the two reviewers manually inspected every single answer. Most models followed a consistent structure: an initial binary judgment ("yes" or "no"), a descriptive rationale, and a concluding summary table indicating which implementations were considered equivalent to the reference. However, models sometimes displayed internal inconsistencies, for example, starting with an affirmative answer, contradicting this position in the explanatory text, and then concluding with a negative assessment (or vice versa). In such cases, when the narrative explanation was coherent with the final summary table, we disregarded the initial contradictory response and considered the final answer.

Particular attention was paid to contradictions, hallucinated justifications, or rapid shifts in stance that signaled unstable reasoning. When a chat session exhibited persistent inconsistencies undermining the interpretability of the explanation, the entire session was discarded and the prompt was resubmitted in a fresh chat. Importantly, no automated post-processing or secondary LLM verification was used: to avoid compounding model errors and to reflect the conditions experienced by real users, the evaluation relied solely on human judgment.

The preamble or contextual priming at the beginning of the prompts follows two distinct formats, illustrated in Fig. 5. Both preambles are zero-shot, as no labeled examples are provided. The contextual preamble combines multiple strategies:

- **Role prompting**: "You are a chatbot for comparing the semantics of small Python programs."
- **Instruction prompting**: "I will provide you with multiple implementations of the same Python function. The first function is the reference version."
- **Contextual information**: "The other functions are perturbed with copy propagation, constant folding, or a combination of the two."

Each of the four prompts was repeatedly submitted to seven chatbots over 10 rounds, with a new chat session initialized for each iteration. This resulted in a total of $11 \cdot 4 \cdot 7 \cdot 10 = 3,080$ outputs (11 Python programs, 4 prompts, 7 chatbots and 10 rounds of experiments). Chatbot responses tend to be highly verbose. To encourage the models' reasoning process, we did not request a simple yes/no answer, as we observed a performance drop in such cases. Each response was carefully analyzed, taking into account potential contradictions in the text and other logical inconsistencies. Rather than post-processing the answers using an additional AI, this preliminary series of experiments relied on human evaluation to mitigate the risk of misinterpreting verbose and complex outputs.

Table 4 Chatbot models and versions involved in our experimentation as of January-February 2025. Experiments have been conducted at various times of the day and over multiple weeks

Company	Product	Model
Github	Copilot Pro	Claude 3.5 Sonnet
Github	Copilot Pro	GPT-4o
Google	Gemini	Gemini 2.0 Flash
OpenAI	ChatGPT	GPT-4o
DeepSeek	DeepSeek	DeepSeek-R1 V3
Amazon Web Services	Q Developer	Various
Anthropic	Claude	Claude 3.5 Sonnet

Table 4 lists the chatbots evaluated in our study together with the model versions available at the time of testing. Premium subscriptions are not included, with the sole exception of Copilot, which requires one. No APIs were used: all experiments were carried out manually through the web interface when available, or via the free Visual Studio Code plugin when that was the only access point (as in the case of Copilot and Amazon Q). This choice is essential for our preliminary analysis, as we aim to assess chatbot performance under the same conditions experienced by the vast majority of casual users. Admittedly, our choice partially undermines the reproducibility and scalability of the results, mainly due to how temperature is set by default in free-to-use versions of the models. Our goal, though, is to examine how the models behave for casual users who typically do not possess premium licences and rely on freely accessible versions. In other words, while the authors are fully aware that most LLM providers offer more capable models through premium plans and/or API usage, we intentionally focus on the baseline implementation, widely accessible to developers worldwide and which does not offer any control on the temperature setting, to probe the intrinsic quality of the models' understanding.

It is worth mentioning that Copilot is not a model on its own, but it serves as a frontend for multiple models; our tests focus on two: ChatGPT and Claude Sonnet. Despite using the same underlying technology as their standalone versions, results on Copilot differ, warranting their inclusion in our experiments. Similarly, Amazon Q reportedly employs Claude 3.5 Sonnet,[1] alongside custom models selected based on the prompt type. However, both Amazon Q and Copilot Claude yield results distinct from their official counterparts.

Additionally, chat-based interaction prevents users from adjusting the *temperature* parameter [8], which controls response variability. Typically ranging from 0 to 1, a temperature of 0 ensures deterministic outputs, while higher values

[1] As of February 2025, the Amazon Q Developer official blog states that Claude Sonnet is the primary model for code review and generation: https://aws.amazon.com/it/blogs/devops/amazon-q-developer-inline-chat.

introduce randomness. Most chatbots default to 0.7, adjusting dynamically, though the exact settings are not always disclosed. Consequently, responses may vary across sessions, times, and even accounts. To account for this variability, our experiments were conducted across different time frames and conditions to ensure a fair sampling of model behavior.

4 Experimental results

Our evaluation is based on a total of 3,080 outputs generated by the seven chatbots under study. For the single-class setting (Prompts 1 and 2), we manually analyzed 1,540 responses, each addressing the semantic equivalence of the three program variants. This yielded 4,620 fine-grained *yes/no* judgments. For the multi-class setting (Prompts 3 and 4), the remaining 1,540 responses each compared a reference program against seven variants, resulting in 10,780 *yes/no* judgments. In total, we manually reviewed all 3,080 chatbot outputs and extracted 15,400 fine-grained equivalence decisions. The aggregated results are presented in Tables 5 and 6.

Table 5 shows the accuracy of the class of semantically equivalent programs across all prompts for each chatbot. The rightmost column shows the average accuracy for each prompt across all chatbots. The bottom row represents the average for each chatbot across all prompts. Prompts 3 and 4 report the accuracy of the class of semantically equivalent programs, despite including both correct and incorrect code. At first glance, Claude Sonnet and DeepSeek emerge as the top performers, with Claude holding a slight edge on average across all prompts (79.09% vs. 76.88%). Conversely, Gemini exhibits the lowest scores in most compartments (43.72% overall), followed somewhat unexpectedly by Copilot Claude (60.19%). Interestingly, despite both purportedly using Claude Sonnet, the model behind Copilot appears to behave differently from Anthropic's implementation. The rest of the models fall in the middle range, with ChatGPT surprisingly not exceeding the average. It is clear that contextual prompts, i.e., Prompts 2 and 4, enhance the accuracy of all chatbots, with the notable exception of Gemini, whose performance unexpectedly declines. The most significant improvement is observed in Copilot ChatGPT and Amazon Q, which appear to benefit greatly from contextual information. Conversely, top-performing models such as DeepSeek and Claude Sonnet show only a minor accuracy gain.

We also investigate the behavior of the chatbots when we insert in the prompt both correct and incorrect perturbations of the reference function (yellow and green rows). In this case, we observe a significant drop in performance compared to the single-class prompt (red and purple rows). Specifically, there is about an 8% drop in average performance between Prompt 1 and Prompt 3, with DeepSeek being the only exception. The same phenomenon happens when we add context, as there is a degradation of about 3.5% on average in performance between Prompt 2 and Prompt 4, with only ChatGPT seeming unaffected.

Finally, we report the overall results of the multi-class prompts for both classes and the partial results. Interestingly, Table 6 exhibits good results in detecting incorrect implementations. In particular, if we compare the two tables, there is no real improvement between the contextless Prompt 3 and the contextual Prompt 4 concerning the class of incorrect programs. However, this increase comes at the expense of the class of semantically equivalent programs. The experiments thus show that there is a classification bias, where the chatbots tend to misclassify correct perturbations when they are also provided with incorrect implementations.

Table 7 provides a detailed breakdown of the results for each type of code perturbation. For each of the 11 algorithms, the accuracy of all chatbots is reported across the three perturbations: copy propagation (CP), constant folding (CF), and their combination (CP + CF). Claude Sonnet and DeepSeek once again stand out, achieving the highest performances even on individual perturbation types. Conversely, Gemini performs worse than the other models overall. On average, constant folding is the least confusing perturbation for chatbots, yielding an accuracy of 76.40%. Copy propagation appears more challenging, with an accuracy of 69.20%, while the combination of both perturbations results in the lowest accuracy, at 49.39%.

5 Discussion

Our investigation into the semantic equivalence of Python programs using Large Language Models (LLMs) has revealed a number of intriguing behavioral patterns and inconsistencies. Our observations highlighted the presence of contradictions, hallucinations, and limited understanding of certain coding patterns. Notably, the models occasionally *contradict themselves* during the evaluation process, stating something at the beginning of the output and landing on the opposite conclusion by the end. For example, an initial affirmative response regarding the equivalence of two functions is sometimes later followed by a retraction or a confusing double negative, a behavior that suggests a superficial rather than robust internal reasoning process. Furthermore, we observed that in many cases, when models initially judged two versions of a program to be non-equivalent (e.g., the "cp" and "cp+cf" perturbed versions), they later stated that the two were equivalent. This flip-flopping not only highlights the internal instability of the models' reasoning but also raises concerns about their ability to maintain logical consistency across the whole output.

Another intriguing observation concerns response *verbosity*. When we instructed models to be concise, their answers were often more erroneous. Though not included in

Table 5 Performance of all chatbots on the class of semantically correct programs across all prompts. Prompts 1 and 2 contain only correct implementations. To ensure a fair comparison, for Prompt 3 and Prompt 4 we report only the performance on the class of correct programs, i.e. those that are truly semantically equivalent. The best and worst performances are highlighted in bold

	VSCode Plugin							
	Copilot		Extension	Web Interface				
Prompt #	Claude	ChatGPT	Amazon Q	Gemini	ChatGPT	DeepSeek	Claude	Average
#1	63.20%	53.68%	56.99%	47.62%	67.53%	70.56%	76.77%	62.34%
#2	71.52%	79.39%	76.36%	41.82%	61.21%	82.42%	**84.85%**	**71.08%**
#3 / Correct Programs	43.03%	51.52%	46.06%	45.45%	46.06%	76.97%	71.72%	**54.40%**
#4 / Correct Programs	63.03%	72.73%	65.45%	**40.00%**	70.91%	77.58%	83.03%	67.53%
Average	60.19%	64.33%	61.22%	**43.72%**	61.43%	76.88%	**79.09%**	

Table 6 Complete results on the classification of both semantically correct and incorrect programs. A comparison is made between prompts with and without contexts

	VSCode Plugin							
	Copilot		Extension	Web Interface				
Prompt #	Claude	ChatGPT	Amazon Q	Gemini	ChatGPT	DeepSeek	Claude	Average
#3 / Correct Programs	**43.03%**	51.52%	46.06%	45.45%	46.06%	76.97%	71.72%	**54.40%**
#3 / Incorrect Programs	94.55%	89.55%	95.00%	**95.91%**	90.00%	95.45%	93.94%	**93.48%**
#3 / Overall	72.47%	73.25%	74.03%	74.29%	**71.17%**	**87.53%**	76.19%	75.56%
#4 / Correct Programs	63.03%	72.73%	65.45%	**40.00%**	70.91%	77.58%	83.03%	**67.53%**
#4 / Incorrect Programs	**98.64%**	85.45%	96.82%	93.64%	90.45%	89.55%	96.36%	**92.99%**
#4 / Overall	83.38%	80.00%	83.38%	**70.65%**	82.08%	84.42%	**90.65%**	82.08%

our statistics, this suggests that extended outputs, reflecting the model's chain of thought, play a crucial role in accurate reasoning. Limiting verbosity may truncate this process, increasing errors—a finding that merits further investigation into the link between output length and reasoning quality in program understanding.

Overall, copy propagation appears to be particularly challenging for most models, with the combination of copy propagation and constant folding being the most problematic case. A closer look at the data suggests this confusion becomes more pronounced when non-numerical data types, such as lists and arrays, are involved. This is likely due to the models' limited understanding of the subtle semantics of Python's *assignment operator*, which behaves differently depending on the data type: numeric types are copied, whereas arrays and lists are merely aliased by reference. In other words, a statement like `B = A`, where `A` is an array, results in a new binding rather than an actual copy of `A`. This distinction appears to be a common source of confusion among the tested models. Notably, only Claude Sonnet and Deepseek demonstrated some ability to differentiate between references and true copies in cases of variable rebinding.

The Sieve of Eratosthenes serves as a particularly revealing example. This algorithm combines arithmetic operations with list manipulation, yielding the lowest accuracy scores across all models. We argue that this poor performance stems from the modulus operation in the if-guard, which appears to confuse most LLMs in determining when the list removal operation takes place in the then-branch. The issue is further exacerbated in perturbed versions of the code, where the presence of copies amplifies the confusion, as models struggle to distinguish between actual list copies and mere references.

Despite the obfuscation of function names, all models rapidly recognized even lesser-known algorithms. Interestingly, the antialiasing algorithm was frequently identified as an "upscaling" technique, even though it also performs downscaling to induce a blur effect. In other words, none of the tested models fully grasped the antialiasing algorithm's dual role – upscaling followed by downscaling – which constitutes a hallmark of its functionality.

Surprisingly, when tested on the 3D point rotation and FFT algorithms, all models achieved a 100% success rate, suggesting that even less canonical snippets are well represented in their extensive knowledge base. Conversely, the

Table 7 Averaged results for each algorithm in the dataset and the corresponding semantically-preserving perturbations

		VSCode Plugin							
		Copilot		Extension	Web Interface				
	Perturb	Claude	ChatGPT	Amazon Q	Gemini	ChatGPT	DeepSeek	Claude	Average
Duplicate Removal	cp	30.00%	85.71%	49.29%	5.00%	86.43%	80.00%	100.00%	62.35%
	cf	82.50%	89.29%	95.00%	40.00%	95.00%	100.00%	100.00%	85.97%
	cp+cf	40.00%	60.71%	44.29%	0.00%	91.43%	100.00%	75.00%	58.78%
Sieve	cp	17.50%	35.00%	5.00%	14.29%	8.57%	18.57%	8.33%	**15.32%**
	cf	35.00%	70.00%	30.00%	3.57%	30.00%	62.14%	100.00%	47.24%
	cp+cf	10.00%	20.00%	0.00%	0.00%	0.00%	8.57%	8.33%	**6.70%**
Count Occurrences	cp	60.00%	100.00%	52.14%	19.29%	62.86%	91.43%	82.50%	66.89%
	cf	92.50%	100.00%	100.00%	67.86%	70.00%	95.00%	100.00%	89.34%
	cp+cf	57.50%	75.00%	52.14%	15.71%	66.43%	95.00%	82.50%	63.47%
Fibonacci	cp	75.00%	58.57%	85.71%	45.71%	47.86%	86.43%	100.00%	71.33%
	cf	87.50%	96.43%	96.43%	47.14%	57.86%	96.43%	100.00%	83.11%
	cp+cf	30.00%	5.00%	37.14%	10.00%	29.29%	27.14%	70.00%	29.80%
Primality	cp	20.00%	33.57%	25.00%	7.14%	62.86%	95.00%	49.17%	41.82%
	cf	87.50%	87.86%	96.43%	72.14%	90.00%	90.00%	100.00%	89.13%
	cp+cf	20.00%	25.00%	5.00%	3.57%	17.14%	15.00%	49.17%	19.27%
Find Duplicate	cp	87.50%	95.00%	95.00%	100.00%	85.00%	100.00%	100.00%	94.64%
	cf	52.50%	77.86%	50.00%	15.71%	44.29%	95.00%	100.00%	62.19%
	cp+cf	37.50%	52.86%	50.00%	10.71%	39.29%	95.00%	100.00%	55.05%
Bubble sort	cp	37.50%	44.29%	37.86%	50.00%	47.86%	58.57%	82.50%	51.22%
	cf	80.00%	92.86%	85.00%	81.43%	96.43%	100.00%	100.00%	90.82%
	cp+cf	37.50%	60.71%	22.86%	46.43%	40.71%	58.57%	82.50%	49.90%
Anti Aliasing	cp	92.50%	70.71%	95.00%	55.71%	81.43%	89.29%	95.83%	82.93%
	cf	30.00%	75.71%	20.00%	69.29%	46.43%	100.00%	86.67%	61.16%
	cp+cf	15.00%	80.71%	10.00%	27.14%	46.43%	70.71%	62.50%	44.64%
FFT	cp	92.50%	100.00%	100.00%	100.00%	95.00%	100.00%	100.00%	98.21%
	cf	92.50%	100.00%	100.00%	100.00%	95.00%	100.00%	100.00%	98.21%
	cp+cf	92.50%	100.00%	100.00%	95.00%	85.00%	100.00%	100.00%	96.07%
Rotate 3D	cp	92.50%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	98.93%
	cf	75.00%	100.00%	100.00%	85.00%	100.00%	100.00%	100.00%	94.29%
	cp+cf	62.50%	85.00%	100.00%	70.00%	100.00%	100.00%	86.67%	86.31%
Unification	cp	87.50%	75.00%	96.43%	75.00%	68.57%	63.57%	76.67%	77.53%
	cf	47.50%	45.00%	46.43%	10.00%	20.00%	22.14%	81.67%	38.96%
	cp+cf	37.50%	45.00%	35.71%	0.00%	20.00%	23.57%	71.67%	33.35%

unification algorithm was consistently the least understood. This is likely due to the nature of the data structures involved: the algorithm computes a substitution – specifically, the Most General Unifier (MGU) – by performing pattern matching over pairs of types. Substitutions are essentially dictionaries, while types are syntactic entities defined through case-based structures in an object-oriented manner. Such a use of classes appears to confuse chatbots, which seem better suited to understanding numerical algorithms, or those involving conventional data structures such as arrays and lists.

5.1 Improving LLMs in code understanding

LLMs' ability to understand code depends on the complexity of the chatbot's model ecosystem, where models may be dynamically selected based on the query type. For instance, ChatGPT appears to use different models/extensions depending on whether the task involves text, images, or code execution.

To assess execution capabilities, we asked chatbots if they had direct access to a Python interpreter. All but two – ChatGPT and Gemini – denied this ability. ChatGPT consistently confirmed that it could execute Python code, although Ope-

nAI provides no official confirmation. Gemini, on the other hand, gave inconsistent answers, sometimes affirming and others denying interpreter access, suggesting variable behavior or conditional access.

In general, techniques for improving performance on code understanding tasks can be grouped into two categories: those that modify the model itself, and those that improve how the model is used, without altering its parameters.

Model-level improvements involve adapting LLMs to reason more robustly about semantic equivalence.

Fine-tuning is a standard approach where a subset of model weights is updated using supervised datasets. This technique can also be applied to LLMs to change a small portion of the model [12, 19], for example, by using Parameter-Efficient Fine-Tuning (PEFT) methods such as Low-Rank Adaptation (LoRA) or adapters. In our case, we could invert the transformation rules outlined in Sect. 3.1 to generate a dataset of semantically equivalent but syntactically varied function implementations. Training on such pairs – both with and without nonequivalent perturbations – could help models learn invariance to superficial code changes. Additionally, contrastive learning [15, 16] could be employed to shape the model's internal representations, bringing equivalent code closer together and pushing non-equivalent code apart.

However, these model-level improvements require access to the model's internal parameters and training procedures, which is typically limited for commercial LLMs (e.g., the full architecture for models of the GPT family is not disclosed). This limitation poses a significant challenge for academic research and practical adaptation, underscoring the importance of collaboration between model developers and the research community to enable specialized fine-tuning for tasks like code understanding.

Usage-level improvements, on the other hand, focus on leveraging existing models more effectively. Prompting is a commonly used and flexible strategy [31]. Basic prompting involves direct instructions, while more advanced forms such as few-shot prompting provide examples to guide the model's behavior. Chain-of-thought prompting, which encourages the model to reason step-by-step through a problem, can notably improve performance on tasks requiring deeper semantic understanding and logical reasoning [20]. Other techniques – such as role-based prompting, where the model is asked to assume a specific persona (e.g., a compiler or code reviewer), or instruction-tuned prompting using clearly structured commands – can also be used effectively in standard chatbot interfaces, without API access. These methods improve task-specific performance without modifying the underlying model.

Another widely used strategy is retrieval-augmented generation (RAG) [13], which enhances LLM outputs by injecting relevant context retrieved from an external source, such as similar code snippets or documentation.

Finally, the complexity of current chatbot ecosystems could readily accommodate the integration of static code analysis or transformation pipelines behind the scenes. If chatbots incorporated a pre-processing phase that normalizes function implementations – using optimizations or rewrites like those described in Sect. 3.1 – they could help the model focus on the semantic content of the code rather than superficial syntactic variation. This type of tool-assisted preprocessing is especially attractive when retraining the model is not feasible.

While these usage-level strategies do not improve the model itself, they often yield substantial gains in practice and offer a practical path forward for applications requiring code understanding.

6 Conclusions

We analyzed LLMs' code understanding by integrating insights from compiler design and advances in generative AI. To generate semantically equivalent code requiring nuanced comprehension, we employed standard compiler optimization techniques: copy propagation and constant folding. In this preliminary study, we defined optimization rules only for Python data types that exhibit call-by-value behaviour. Consequently, mutable structures such as lists and arrays, whose operational semantics are effectively call-by-reference, are excluded. Also, since these optimizations are typically defined on control flow graphs of intermediate representations, we adapted them to work directly at the source code level.

Our experiments highlight both the strengths and limitations of current LLMs in code understanding. While contextual prompting can improve performance in some cases, it also reveals inconsistencies in logical reasoning. Results indicate that zero-shot contextless queries yield a 59% accuracy, while adding contextual information improves to a 71%. Despite a 12% increase, a 29% error rate is still too high to broadly trust these models for code reasoning. We suggest that a code pre-processing tool integrated into the LLM pipeline would improve program understanding by a fair amount.

Future work should disentangle these effects, refining both LLM reasoning and evaluation methods for complex semantic tasks. Further directions include expanding the dataset, testing additional models, and exploring more advanced optimizations.

While usage-level improvements such as prompting and preprocessing can provide practical performance gains, our study underscores the necessity of adapting the LLMs themselves to robustly handle semantic code equivalence. Such model-level modifications, including fine-tuning or contrastive learning on semantically equivalent code, require direct access to and control over the models' parameters.

Given the proprietary nature of the commercial LLMs evaluated here, only their owners can realistically implement these changes. This highlights a critical role for developers of large language models to facilitate or enable specialized fine-tuning and adaptation to advance code understanding capabilities.

Acknowledgements This study was carried out within the PE0000014 - Security and Rights in the CyberSpace (SERICS) and received funding from the European Union Next-GenerationEU - National Recovery and Resilience Plan (NRRP) – MISSION 4 COMPONENT 2, INVESTIMENT 1.3 – CUP N. H73C22000890001. This work has been also partially supported by the Research Project INDAM GNCS 2024 - CUP E53C23001670001 "Modelli composizionali per l'analisi di sistemi reversibili distribuiti (MARVEL)" and by the Project PRIN 2020 - CUP N. 20202FCJMH "NiRvAna - Noninterference and Reversibility Analysis in Private Blockchains". This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Funding information Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Boston, MA, USA (2006)
2. Amazon website Services: Amazon q developer website, https://aws.amazon.com/it/q/developer/, accessed: 2025-02-21
3. Aydin, F., Aysu, A.: Leaking secrets in homomorphic encryption with side-channel attacks. J. Cryptogr. Eng., 1–11 (2024)
4. Cao, Y., Li, S., Liu, Y., Yan, Z., Dai, Y., Yu, P.S., Sun, L.: A comprehensive survey of ai-generated content (aigc): a history of generative ai from gan to chatgpt (2023). arXiv:2303.04226. arXiv preprint
5. Chandrasekaran, D., Mago, V.: Evolution of semantic similarity—a survey. ACM Comput. Surv. **54**(2), 1–37 (2021)
6. Chatgpt website. https://github.com/features/copilot, accessed: 2025-02-21
7. Dakhel, A.M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., Jiang, Z.M.J.: Github copilot ai pair programmer: asset or liability? J. Syst. Softw. **203**, 111734 (2023)
8. Davis, J., Van Bulck, L., Durieux, B.N., Lindvall, C., et al.: The temperature feature of chatgpt: modifying creativity for clinical research. JMIR Human Factors **11**(1), e53559 (2024)
9. Dolcetti, G., Arceri, V., Iotti, E., Maffeis, S., Cortesi, A., Zaffanella, E.: Helping llms improve code generation using feedback from testing and static analysis (2024). arXiv:2412.14841. arXiv preprint
10. Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al.: The llama 3 herd of models (2024). arXiv:2407.21783. arXiv preprint
11. Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., Tsang, R., Nazari, N., Wang, H., Homayoun, H., et al.: Large language models for code analysis: do {LLMs} really do their job? In: 33rd USENIX Security Symposium (USENIX Security, vol. 24, pp. 829–846 (2024)
12. Han, Z., Gao, C., Liu, J., Zhang, J., Zhang, S.Q.: Parameter-efficient fine-tuning for large models: a comprehensive survey (2024). arXiv:2403.14608. arXiv preprint
13. He, P., Wang, S., Chowdhury, S., Chen, T.H.: Evaluating the effectiveness and efficiency of demonstration retrievers in rag for coding tasks. In: 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 500–510. IEEE, New York (2025)
14. Henkel, J., Ramakrishnan, G., Wang, Z., Albarghouthi, A., Jha, S., Reps, T.: Semantic robustness of models of source code. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 526–537. IEEE, New York (2022)
15. Hu, H., Wang, X., Zhang, Y., Chen, Q., Guan, Q.: A comprehensive survey on contrastive learning. Neurocomputing **610**, 128645 (2024)
16. Jaiswal, A., Babu, A.R., Zadeh, M.Z., Banerjee, D., Makedon, F.: A survey on contrastive self-supervised learning. Technologies **9**(1), 2 (2020)
17. Jiang, A.Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D.S., Casas, D.D.L., Hanna, E.B., Bressand, F., et al.: Mixtral of experts (2024). arXiv:2401.04088. arXiv preprint
18. Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large language models are zero-shot reasoners. Adv. Neural Inf. Process. Syst. **35**, 22199–22213 (2022)
19. Latif, E., Zhai, X.: Fine-tuning chatgpt for automatic scoring. Comput. Educ. Artif. Intell. **6**, 100210 (2024)
20. Li, J., Li, G., Li, Y., Jin, Z.: Structured chain-of-thought prompting for code generation. ACM Trans. Softw. Eng. Methodol. **34**(2), 1–23 (2025)
21. Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., Zhang, L., Li, Z., Ma, Y.: Exploring and evaluating hallucinations in llm-powered code generation (2024). arXiv:2404.00971. arXiv preprint
22. Maroengsit, W., Piyakulpinyo, T., Phonyiam, K., Pongnumkul, S., Chaovalit, P., Theeramunkong, T.: A survey on evaluation methods for chatbots. In: Proceedings of the 2019 7th International Conference on Information and Education Technology, pp. 111–119 (2019)
23. OpenAI: Openai chatgpt website, https://openai.com/blog/chatgpt, accessed: 2025-02-21
24. Pellicer, L.F.A.O., Ferreira, T.M., Costa, A.H.R.: Data augmentation techniques in natural language processing. Appl. Soft Comput. **132**, 109803 (2023)
25. Pour, M.V., Li, Z., Ma, L., Hemmati, H.: A search-based testing framework for deep neural networks of source code embedding. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 36–46. IEEE, New York (2021)
26. Rabin, M.R.I., Alipour, M.A.: Evaluation of generalizability of neural program analyzers under semantic-preserving transformations (2020). arXiv:2004.07313. arXiv preprint
27. Ressi, D., Romanello, R., Piazza, C., Rossi, S.: Neural networks reduction via lumping. In: International Conference of the Italian Association for Artificial Intelligence, pp. 75–90. Springer, Berlin (2022)
28. Ressi, D., Romanello, R., Piazza, C., Rossi, S.: Ai-enhanced blockchain technology: a review of advancements and opportunities. J. Netw. Comput. Appl., 103858 (2024)

29. Ressi, D., Romanello, R., Rossi, S., Piazza, C.: Compressing neural networks via formal methods. Neural Netw., 106411 (2024)
30. Ressi, D., Spanò, A., Benetollo, L., Piazza, C., Bugliesi, M., Rossi, S.: Vulnerability detection in Ethereum smart contracts via machine learning: a qualitative analysis (2024). arXiv:2407.18639. arXiv preprint
31. Sahoo, P., Singh, A.K., Saha, S., Jain, V., Mondal, S., Chadha, A.: A systematic survey of prompt engineering in large language models: Techniques and applications (2024). arXiv preprint. arXiv:2402.07927
32. Spanò, A.: Github repository. https://github.com/alvisespano/perturb
33. Spracklen, J., Wijewickrama, R., Sakib, A.N., Maiti, A., Viswanath, B.: We have a package for you! A comprehensive analysis of package hallucinations by code generating {LLMs}. In: 34th USENIX Security Symposium (USENIX Security, vol. 25, pp. 3687–3706 (2025)
34. Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M.S., Love, J., et al.: Gemma: Open models based on Gemini research and technology (2024). arXiv preprint. arXiv:2403.08295
35. Urban, C., Miné, A.: A review of formal methods applied to machine learning (2021). arXiv:2104.02466. arXiv preprint
36. Wermelinger, M.: Using github copilot to solve simple programming problems. In: Proceedings of the 54th ACM Technical Symposium on Computer Science Education V, vol. 1, pp. 172–178 (2023)