Toward Explaining Large Language Models in Software Engineering Tasks

Toward Explaining Large Language Models in Software Engineering Tasks

ANTONIO VITALE*, University of Molise & Politecnico di Torino, Italy
KHAI-NGUYEN NGUYEN*, William & Mary, USA
DENYS POSHYVANYK, William & Mary, USA
ROCCO OLIVETO, University of Molise, Italy
SIMONE SCALABRINO, University of Molise, Italy
ANTONIO MASTROPAOLO, William & Mary, USA

Recent progress in Large Language Models (LLMs) has substantially advanced the automation of software engineering (SE) tasks, enabling complex activities such as code generation and code summarization. However, the black-box nature of LLMs remains a major barrier to their adoption in high-stakes and safety-critical domains, where explainability and transparency are vital for trust, accountability, and effective human supervision. Despite increasing interest in explainable AI for software engineering, existing methods lack domain-specific explanations aligned with how practitioners reason about SE artifacts. To address this gap, we introduce `FeatureSHAP`, the first fully automated, model-agnostic explainability framework tailored to software engineering tasks. Based on Shapley values, `FeatureSHAP` attributes model outputs to high-level input features through systematic input perturbation and task-specific similarity comparisons, while remaining compatible with both open-source and proprietary LLMs. We evaluate `FeatureSHAP` on two bi-modal SE tasks: code generation and code summarization. The results show that `FeatureSHAP` assigns less importance to irrelevant input features and produces explanations with higher fidelity than baseline methods, at a substantially lower computational cost than token-level alternatives. A practitioner survey involving 37 participants shows that `FeatureSHAP` helps practitioners better interpret model outputs and make more informed decisions. Collectively, `FeatureSHAP` represents a meaningful step toward practical explainable AI in software engineering. `FeatureSHAP` is available at https://github.com/deviserlab/FeatureSHAP.

CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: Large Language Models, Code Generation, Code Summarization, Explainability, Feature Attribution, Trustworthy AI

1 Introduction

Automation in software engineering (SE) is rapidly evolving due to two main drivers: the unprecedented volume and sheer amount of code data available from open-source ecosystems such as GitHub and the rise of Large Language Models (LLMs) capable of understanding and generating code [15, 27, 30, 32, 44, 62]. Among these,

*Both authors contributed equally to this research.

Authors' addresses: Antonio Vitale, University of Molise & Politecnico di Torino, Termoli, Italy, antonio.vitale@unimol.it, antonio.vitale@polito.it; Khai-Nguyen Nguyen, William & Mary, Williamsburg, Virginia, USA, knguyen07@wm.edu; Denys Poshyvanyk, William & Mary, Williamsburg, VA, USA, dposhyvanyk@wm.edu; Rocco Oliveto, University of Molise, Italy, rocco.oliveto@unimol.it; Simone Scalabrino, University of Molise, Italy, simone.scalabrino@unimol.it; Antonio Mastropaolo, William & Mary, Williamsburg, VA, USA, amastropaolo@wm.edu.

ACM 1557-7392/2026/8-ART
https://doi.org/10.1145/3837084

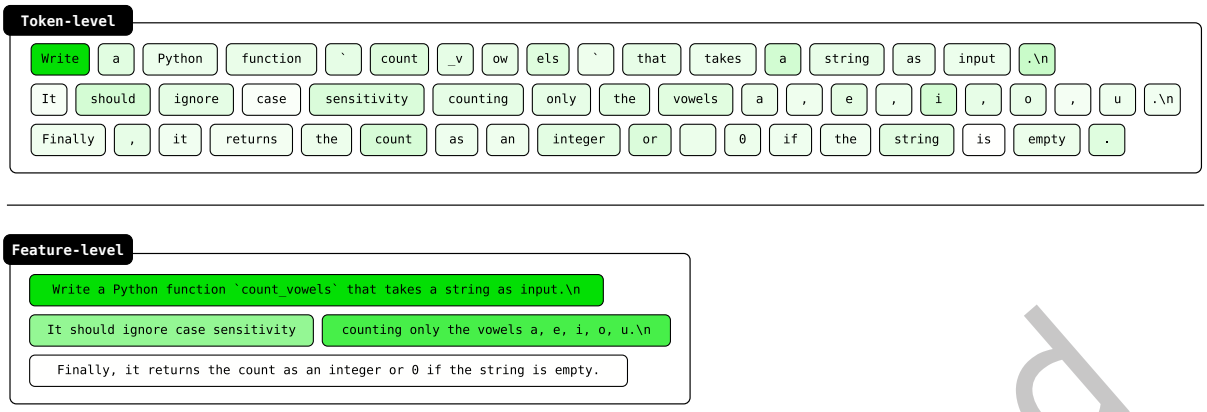

Fig. 1. Token-level (top) and feature-level (bottom) attributions for the same input; darker rectangles represent higher attribution scores.

Neural Code Models (NCMs) represent a focused branch of LLM research–models specifically optimized for coding-related tasks, offering finer control and domain awareness in SE applications [21, 25, 26, 66, 74]. Trained in a vast corpus of text enriched with large-scale open-source code, these models have achieved impressive performance in a wide range of software engineering applications, including code completion [15, 16], program repair [19, 35, 36, 79], code summarization [5, 7, 69, 73], and test case generation [8, 39, 63]. Through scaling to billions and even trillions of parameters[1], LLMs are capable of capturing complex relationships between code and natural language, allowing them to generalize effectively across programming languages, coding styles, and diverse development contexts. This capability has placed LLMs at the core of commercial software engineering tools, such as GitHub Copilot [23], which promise to enhance developer productivity throughout the software development life-cycle [52, 53, 70]. This signals a fundamental shift in SE automation, where AI-driven models are no longer confined to experimental settings, but are actively embedded in IDEs and integrated into development workflows.

However, LLMs operate as black boxes. Given a suggestion (*e.g.,* a generated code block), a developer can either blindly trust the model, which actually results in a productivity improvement, or not, which implies double-checking the output and, thus, spending more time on the task. In addition, LLMs are non-deterministic, *i.e.,* the same input can result in different outputs. Such challenges pose significant barriers to the trustworthy adoption of such models. Ideally, developers should neither blindly trust the model – because of non-determinism – nor completely mistrust it – since it would dramatically reduce the benefits of using LLMs in the first place. Instead, we conjecture that the best option would be an *evidence-based trust*: predictions should be accompanied by easy-to-check pieces of evidence that report to what extent the user can trust the output and its parts. Explainable AI (xAI) for SE methods emerge as a practical and promising solution to this problem. xAI methods reveal which inputs most influence an LLM's output, enabling practitioners to better assess the reliability of generated code, debug unexpected behavior, and refine prompts to align the model with the user's intended goals. Among existing xAI techniques, two of the most popular are LIME [61] and SHAP [46]. LIME [61] explains a prediction by generating perturbed versions of the input (*e.g.,* by masking or removing tokens) and training a simple interpretable model to approximate how these perturbations affect the generated output. SHAP [46], instead, estimates the importance of each input token by measuring how the inclusion or exclusion of that element

[1]For example, recent large-scale models such as OpenAI's GPT, Anthropic's Claude, and Google's Gemini are reported to contain on the order of one to several trillion parameters, depending on the configuration disclosed.

changes the output across all possible subsets of tokens. Existing explainability techniques typically operate at the level of individual tokens, which limits their applicability in software engineering contexts. For example, consider the attributions for the prompt shown in Fig. 1. In the token-level view, the attribution is spread across many tokens, resulting in an explanation that is difficult to interpret: Developers do not reason one token at a time, and the distribution of importance across subwords offers little insight into which conceptual parts of the instruction actually contributed to model's output. Instead, the feature-level view provides a different perspective. Grouping tokens into semantically coherent units, such as the task description, the constraint on case sensitivity, and the definition of the vowel set, produces an explanation that aligns more closely with how developers understand and reason about prompts. This higher-level representation makes the explanation substantially more actionable. For instance, when a specific feature receives negligible attribution (*e.g.,* the final requirement in the example), the developer may immediately infer two possibilities: Either the model ignored an intended constraint, which may imply a potential issue in the generated code, or the model produced the correct result regardless, explaining that the overlooked requirement was not contributing in the first place (*i.e.,* the requirement could have been omitted without affecting the model's behavior). In both cases, feature-level attribution provides a clearer, more practical basis for explaining the model's behavior than token-level analysis and naturally points toward the need for xAI techniques that operate over semantically meaningful, task-dependent *features*—units. When the input consists of source code, such units may align with program structures, such as method signatures, code blocks, documentation fragments, that define the semantics developers rely on when reasoning about the way the model processed and responded to the input.

To this end, we introduce `FeatureSHAP`, a model- and task-agnostic explainability framework that estimates the importance of an input i in generating the related output $M(i)$. Grounded in the theoretical foundations of Shapley values [64], `FeatureSHAP` provides a principled mechanism for estimating feature attribution beyond specific model architectures. Unlike recent efforts such as do_{code} [57], AST_{rust} [56], and *CodeQ* [55], which focus on interpretability, `FeatureSHAP` advances explainability by attributing model outputs to semantically meaningful, feature-level code constructs rather than token-level elements, producing explanations that better align with how developers reason about software artifacts.

`FeatureSHAP` follows a three-step workflow. First, it decomposes the input into semantically meaningful units, termed *features*, which capture higher-level abstractions. Second, it performs a series of controlled perturbation-based simulations, systematically including or excluding each feature or combination of features. Third, by observing the corresponding changes in the output of the model, `FeatureSHAP` quantifies the contribution of each feature, producing attributions (a real value score in the range $[0, 1]$) that explain how and to what extent the output depends on different parts of the input. Higher attributions assigned to features indicate a greater influence on the output.

To capture the breadth of the output produced by LLMs, we evaluated `FeatureSHAP` on two representative tasks: code generation and code summarization. Together, they span the spectrum from the production of executable code to the generation of technical natural language, illustrating the generality of our approach.

In particular, we first try to understand whether `FeatureSHAP` can detect parts of the prompt that surely do not influence the output. To this end, given a model M, we consider a set of prompts I, infer the results $O = \{M(i) \ \forall \ i \in I\}$, and then introduce extraneous features to the results $I_n = \{\text{inject}(i, ext(i)) \ \forall \ i \in I\}$. Such features ($ext(i)$) are chosen so that they are surely non-influential, *i.e.,* by making sure that the output of the model does not change ($O_n = \{M(i) \ \forall \ i \in I_n\} = O$). By construction, a reliable xAI approach should attribute a very low score to such extraneous features. Our results show that `FeatureSHAP` consistently assigns negligible attribution scores to injected irrelevant features across all models and tasks, outperforming both random, LLM-based attribution, and token-level explainability baselines in distinguishing non-influential inputs.

In the second investigation, we examine whether `FeatureSHAP` can provide human-centered and actionable support to developers by providing transparent explanations. To this end, we conducted a survey with 37

professionals who regularly employ LLMs in their daily programming activities. Participants were asked to assess FeatureSHAP explanations in code generation and summarization tasks, evaluating their faithfulness and usefulness to support practical use. Our results indicate that practitioners considered the attributions produced by FeatureSHAP meaningful, enhancing their confidence and decision-making when assessing model generated outputs. The combined findings of such two studies confirm both the technical reliability and the practical utility of FeatureSHAP for explainable AI in software engineering.

Finally, we assess the practical cost of adopting FeatureSHAP by measuring both its execution time and monetary cost, and comparing them against token-level explainability alternatives. Our results indicate that FeatureSHAP can generate explanations within a few seconds while requiring substantially less time and lower cost than token-level alternatives.

To summarize, this work advances the state-of-the-art in explainable AI methods for SE via the following key contributions:

- It proposes the first fully automated, model-agnostic, and black-box explainability framework explicitly designed for software engineering tasks.
- It provides empirical evidence supporting the usefulness of FeatureSHAP in (i) finding irrelevant features in the prompt, (ii) helping developers take decisions on the received recommendations, and (iii) reducing execution time and monetary cost compared with token-level explainability alternatives.

In the remainder of this paper, we begin by providing the necessary background to help readers better understand the details of our proposed technique. We then introduce FeatureSHAP, detailing the key components specifically designed to enable concept-level (feature-wise) explainability. Sec. 4 describes the experimental design, while Sec. 5 presents and analyzes our findings through both quantitative metrics and a qualitative user study. In Sec. 6, we situate our work within the broader landscape of explainable and interpretable software engineering, and reflect on the new opportunities unlocked by FeatureSHAP. We conclude by discussing the threats to validity associated with our study and summarizing the main takeaways.

All code and data used in our study are publicly available [1], together with the associated tool at https://github.com/deviserlab/FeatureSHAP.

2 Background & Related Work

We first discuss the background of the concepts of explainability and interpretability. Then, we present related works specifically in the context of software engineering.

2.1 Explainability and Interpretability

There are two families of methodologies that can be adopted to make LLMs less opaque: interpretable methods and explainable methods [47, 50, 81]. Interpretable methods are those where the model or its features can be directly inspected by humans. A classic example is linear regression, where model coefficients explicitly quantify the contribution of each input feature to the prediction. By extension, in the SE domain, an interpretable method might involve the examination of specific parts or features of the models. For example, Sharma et al. [65] examined the Transformer's [71] [2] attention weights in a code-pretrained BERT [18] model to identify which categories of tokens (*e.g.,* identifiers, separators) receive the most attention, suggesting their importance for the model's predictions. In essence, an interpretable method aids in understanding the input–output relationship when certain conditions are met: (i) the model is transparent, meaning its internal structure and parameters can be directly examined; and (ii) the model or its features exhibit traceable behavior, such that specific patterns in the input can be logically linked to corresponding outputs through an understandable decision process.

[2]Transformers are a deep learning architecture based on self-attention mechanisms, which power modern LLMs due to their ability to capture long-range dependencies and represent complex relationships in sequential data such as natural language and source code.

Explainable methods refer to *post-hoc* techniques that take a complex black-box model and produce an explanation for its outputs. Notably, they place no requirement for the model to be open-source or internally accessible-making them particularly valuable when working with proprietary or API-based systems. Common general techniques include LIME [61] and SHAP [46] that follow this paradigm by perturbing input features and observing corresponding changes in output to estimate the importance of features.

Given the focus of our paper on *explainability* rather than *interpretability*, we will report on the post-hoc explainability techniques that have been proposed in the literature.

LIME (Local Interpretable Model-agnostic Explanations) [61], focuses on explaining why a model provides a particular output given a single input leveraging an interpretable model that learns to mimic the model's behavior when exposed to small perturbations of that input. It works by creating a set of synthetic inputs that are slight variations of the original one. For example, in the context of text generation, it randomly removes or alters some tokens. Then, these perturbed inputs are given to the model to quantify how the generation changes. Let us consider the case in which a prompt asks a model to generate code to sort a list: If removing the word `{sort}` from the prompt causes the model to generate something completely different, LIME will infer that `{sort}` was important to generate the original output. Repeating this procedure through many perturbed examples and their corresponding predictions, LIME fits a simple, interpretable model (such as a linear model) with these data points. The weights learned by the surrogate model explain which parts of the original input were most important in steering towards the original generation.

SHAP (SHapley Additive exPlanations) [46] is based on cooperative game theory. SHAP assigns each input feature an importance value based on how much it contributes, on average, to the prediction, measured considering all possible combinations in which tokens could be added to the model input. Intuitively, the model can be imagined as a team project, and each input feature can be imagined as a team member. SHAP evaluates how the final outcome would change as each team member is part of the group, averaging all possible orders in which they could be included. For example, if a code prompt contains both `{sort}` and `{reverse}`, SHAP asks: "What value does adding `{sort}` provide to the model's output, when considered alongside every possible subset of the other features?" By systematically averaging these contributions, SHAP assigns each feature an overall importance score. SHAP is the only method that provides additive feature attributions while guaranteeing local accuracy, missingness, and consistency properties. However, calculating the exact SHAP values requires evaluating the model on every possible subset of input tokens, which becomes infeasible as the input size grows.

Recently, to meet LLMs in the explainability context, TokenSHAP [29] has been proposed. Like SHAP, TokenSHAP is grounded in cooperative game theory, treating each token as a "player" and estimating how much each contributes to the model's output. The intuition is straightforward: By systematically removing tokens and observing how the model's response changes, TokenSHAP can estimate the average effect that each token has on the original generation. Since evaluating every possible combination of tokens is computationally infeasible for real-world prompts, TokenSHAP employs a Monte Carlo sampling strategy to approximate the Shapley value for each token efficiently. For a given prompt, it generates many variants of the prompt by omitting different tokens, queries the LLM for each, and measures how similar each response is to the full prompt TF-IDF representations of the outputs. The average change in response similarity, when a specific token is included versus excluded, determines the token importance score. For example, if the word `{sort}` is removed from a code generation prompt and the model does not suggest (again) sorting operations, TokenSHAP will assign a high importance to `{sort}` token.

Building on token-level approaches, SyntaxSHAP [9] has been proposed to incorporate higher-level linguistic structure into explanations. Like SHAP, it is grounded in cooperative game theory, but instead of forming arbitrary subsets of tokens, SyntaxSHAP constrains coalitions to follow the dependency parse tree of the input text. This ensures that related tokens are treated as coherent units. For instance, given the prompt `{sort the list in descending order}`, SyntaxSHAP evaluates the phrase `{descending order}` as a single unit when estimating

its contribution to the output, moving beyond token-based attributions and produces explanations that are more syntactically faithful and coherent. However, the reliance on surface syntax means that the explanations still diverge from human expectations when semantic concepts are overlooked, leaving open the broader challenge of aligning explanations with domain-specific reasoning needs. SyntaxSHAP also relies on natural language dependency parsing and explains individual next-token predictions, making it inapplicable to source code inputs and to whole-output attribution in black-box settings.

2.2 Explainability and Interpretability in Software Engineering

In software engineering, efforts toward making neural models more transparent have grown alongside the adoption of large language models for code-related tasks. Early work on model transparency in SE primarily emphasized interpretability–that is, understanding how models make predictions–rather than explainability, which focuses on why specific predictions occur and how they relate to human reasoning processes [10].

Among recent contributions, Palacio et al. introduced several interpretability frameworks for neural code models. Their work on do_{code} [57] proposes a post-hoc interpretability approach based on causal inference, designed to reveal how specific code properties influence model decisions. Through case studies on diverse deep learning architectures, they show that do_{code} can expose model sensitivities to syntactic variations and help identify potential confounding factors in code representations.

In the follow-up work, the same authors presented AST_{rust} [56], which explains the model predictions by associating confidence values with syntactic structures in Abstract Syntax Trees (ASTs). Unlike token-based saliency methods, AST_{rust} grounds explanations in language-specific constructs, bridging model behavior, and programming syntax.

Most recently, *CodeQ* [55] was proposed as a rationale-based interpretability method that identifies minimal input subsets–termed code rationales–responsible for model predictions. Through exploratory analysis and a user study, *CodeQ* was shown to produce informative, human-readable explanations and facilitate comparisons between model and developer reasoning.

Collectively, these works have advanced interpretability for neural code models, yet they share important limitations. Each approach requires access to internal model states or token-level confidence scores, restricting their applicability to black-box or proprietary models and making their performance dependent on model calibration. Furthermore, most existing techniques remain anchored at the token or sub-token level, yielding dense visualizations that highlight low-level text fragments rather than semantically meaningful code structures such as function bodies, documentation blocks, or API calls. As a result, their explanations often diverge from the way developers conceptualize and review software artifacts.

Finally, the metrics commonly used to quantify feature importance–such as output-probability shifts or surrogate-model weights–are largely domain-agnostic and overlook the semantics of software engineering tasks.

Moreover, token-level attributions are inherently difficult to interpret and reason about, often resulting in dense and fragmented heatmaps that obscure rather than clarify model behavior [14]. To address this limitation, `FeatureSHAP` introduces an abstraction layer over fine-grained token attributions, aggregating them into higher-level features that correspond to semantically meaningful program elements or documentation components. This abstraction improves readability and bridges the gap between low-level model behavior and the conceptual structures that practitioners naturally employ when reasoning about software systems. Furthermore, `FeatureSHAP` is a model-agnostic explainability technique that operates only on observable input–output behavior, allowing the generation of explanations even when model internals are inaccessible. In doing so, it broadens applicability to proprietary or black-box systems and strengthens trust in real-world production-level environments. Collectively, these characteristics highlight the need for explainability approaches such as `FeatureSHAP`–methods that function

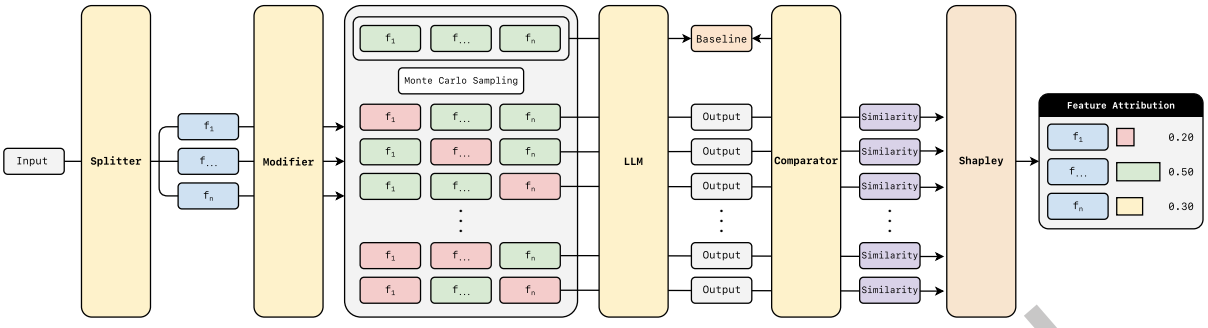

Fig. 2. FeatureSHAP pipeline.

at a higher semantic level, remain model-agnostic, and align with how developers naturally understand and interact with software.

3 FeatureSHAP

To overcome the core limitations of state-of-the-art techniques, we propose FeatureSHAP, a flexible, black-box approach designed specifically for software engineering tasks. Built on the foundations of TokenSHAP [29], FeatureSHAP goes beyond simple token attributions by allowing the input to be partitioned into semantically meaningful features *i.e.,* documentation blocks or code sections, that better align with how developers naturally interpret code. This shift from treating each token as a "player" in the Shapley framework to grouping tokens into higher-level, domain-relevant features enables explanations that are both more interpretable and computationally efficient. As a result, FeatureSHAP provides fine-grained task-adaptable attributions that better match the real-world reasoning of software engineering practitioners.

The definition of features is not fixed a priori. In general, such features can be adapted at runtime depending on the task or developer needs (*e.g.,* grouping prompt elements into high-level requirements or finer-grained constraints). This flexibility is, however, bounded by a set of constraints: features must form a partition of the input, preserve the original content without modification, and correspond to coherent units with respect to the task. These conditions ensure that, despite the flexibility in feature design, the resulting attribution process remains well-defined within the Shapley framework. Therefore, features constitute the players of the Shapley game, replacing individual tokens as the atomic units over which marginal contributions are computed.

From a high-level perspective, FeatureSHAP takes a given input I, a model M, and the output generated by the model given the input $O = M(I)$, and returns (i) the *features*, *i.e.,* a partition of the tokens identified in I ($F_i \in \mathcal{P}(\text{tokenize}(I))$, and (ii) an *attribution function* for each of them $\alpha : f \rightarrow \mathfrak{R}$. The higher the attribution assigned to a given feature ($\phi(f \in F_i)$), the more important the feature is for the model M to generate O from I.

The FeatureSHAP approach consists of three main components: *splitter*, *modifier*, and *comparator*. Each of these can be adapted or instantiated to fit the demands of a particular task at hand. In the following, we first present the roles of the three components and then explain the process followed by FeatureSHAP to compute attribution values for each feature.

The Feature Attribution Process in FeatureSHAP. In Fig. 2 we report on the inner working of FeatureSHAP. Given an input $I = \langle t_1, t_2, \ldots, t_m \rangle$, where t_m is the m-th token, the process begins by passing I to the *splitter* module. Such a module determines how the input is partitioned into features. Formally, the splitter is a function σ that, given a sequence of tokens extracted from the input I (tokenize(I)), returns a specific partition of this sequence, *i.e.,* $\langle f_1, \ldots, f_n \rangle$, where each f_i is a subsequence of tokens in I. The splitter could theoretically be as simple

as a line or sentence splitter, but ideally it should identify a partition of tokens (features) so that each feature is *semantically* meaningful to a developer. Once features are identified, the *modifier* module implements how the input is altered to assess the impact of some specific feature, by mirroring the canonical removal step in the Shapley value framework. Modifier modules are based on an alteration function μ that, given a feature f_i resulting from the use of the splitter module, returns an altered version of such a sequence, f_i'. The simplest alteration function is one that always removes the feature from the input *i.e.,* $\mu_r(f) = \langle\rangle$. Ideally, the modifier module should alter all the possible combinations of features ($\mathcal{P}(\sigma(I))$). Since this is very expensive in practice, to efficiently cover the space of all possible feature subsets, we use Monte Carlo sampling [51] as in TokenSHAP [29]. For each chosen set of features to alter, the modifier module alters them and leaves the others as in the original input. For example, given the input I and its features $\sigma(I) = \langle f_1, f_2, f_3, f_4\rangle$, if the sample includes two sets of features to be altered, *i.e.,* $\{\{f_2\}, \{f_3, f_4\}\}$, the modifier will produce two altered versions of I, *i.e.,* $I_1' = \langle f_1, \mu(f_2), f_3, f_4\rangle$ and $I_2' = \langle f_1, f_2, \mu(f_3), \mu(f_4)\rangle$. Finally, the *comparator* module defines the value function for the Shapley game, *i.e.,* measures how the model's output changes when the altered inputs I_j' are used instead of the original ones I. Formally, the comparator is a similarity function s that, given the output before the alteration ($M(I)$) and the one obtained with one of the altered versions I_j' ($M(I_j')$), measures to what extent they are close. Two completely different outputs will have score 0, while identical ones will have score 1. An example of a comparator function is the classic TF-IDF cosine similarity used in previous work [29]. All computed similarity scores are collected and fed to the *Shapley engine* that estimates the contribution of each feature f_i. By analyzing how similarity changes when features are included or omitted in different combinations, the Shapley engine produces a set of attribution scores $(\phi_1, \ldots, \phi_m)$ for each feature.

Instantiation of `FeatureSHAP`. In this work, we instantiate `FeatureSHAP` by making specific design decision in terms of the *splitter*, *modifier* (specifically, the altering function), and *comparator* components for the tasks we aim to tackle in the experiment, as we will explain later, *i.e.,* code generation and code summarization. First, since the *splitter* strictly depends on the input type and even task at hand, we defined two different splitters: one for tasks that require natural language text as input (*e.g.,* code generation) and one for tasks that require source code as input (*e.g.,* code summarization).

As for the former σ_n, we use a few-shot in-context learning approach where an LLM splitter segments input prompts into high-level features such as summary, detailed description, and the function signature, based on human-annotated examples. We use `gpt-4.1-mini` [54] as the engine behind the LLM splitter to balance both performance and cost. We use a prompt engineered to ensure the high quality outputs described in Fig. 3. In detail, we enforce a rigid output format: the LLM must return a JSON object where each key is a string index ("0", "1", ...) and each value is a feature representing an exact substring of the original docstring, including all whitespace and formatting. We constrain the model to produce outputs that are lossless with respect to the input, thus preventing paraphrasing and ensuring that the segmented units can be leveraged as structured features. We do this by verifying that the ordered concatenation of all extracted segments reproduces the original docstring exactly. A demonstration of the ICL examples used by the splitter can be observed in Fig. 4.

As for the splitter that treats the source code σ_c, we leverage the Tree-Sitter parser[3]. First, we build the abstract syntax tree (AST) of the method, denoted $\mathcal{T}(i)$. Then, since the parser operates on the UTF-8 encoded form of the snippet, we use byte offsets to identify the relevant code blocks. From $\mathcal{T}(i)$, we extract the method declaration (s_0) together with the ordered list of its top-level statements $(s_1, s_2, \ldots, s_k)$. Each of these nodes contains a starting byte position $p(s_i)$ in the encoded input i. We collect all such positions, including the end-of-input position $p(s_{k+1})$, sort them in ascending order, and produce the i-th block by slicing the encoded text from $p(i_j)$ to $p(s_{i+1})$. Formally, $\sigma_c(i)$ returns the features $\langle f_0, \ldots, f_k\rangle$ where $f_i = \text{slice}\big(i,\ p(s_i),\ p(s_{i+1})\big)$ for $i = 0, \ldots, k$.

[3]https://tree-sitter.github.io/tree-sitter/

Prompt

SYSTEM_PROMPT: You are a docstring segmenter. Split the user given docstring into granular sections.

CRITICAL REQUIREMENTS:

- You MUST always output a list of strings - no other format is acceptable
- You MUST NOT paraphrase, rewrite, or modify any text - only extract exactly as written
- Each string must be an exact substring from the original docstring
- Include all original whitespace, newlines, and formatting exactly as they appear

JSON FORMATTING REQUIREMENTS:

- Output ONLY valid JSON - an object with numeric keys and string values: {``0'': ``string1'', ``1'': ``string2'', ``2'': ``string3''}
- Each string value must be a meaningful coding sentence segments.
- Every key must be a string representing the index: "0", ``1'', ``2'', etc.
- Every value must be a string segment enclosed in double quotes
- Ensure all quotes and special characters are properly escaped

Here are examples, your response should follow this format:
{ICL Example 1}
{ICL Example 2}
{ICL Example 3}

USER_INPUT: Docstring: {docstring}

Fig. 3. Prompt for the LLM-based splitter (code generation). Here, {docstring} refers to the docstring-formatted prompt used by BigCodeBench for code generation.

ICL Example 1

INPUT:
"Search for a specific string within the JSON data of files in a given directory and its subdirectories. This function recursively scans the specified directory for JSON files, then checks each file to see if the given string is present within the JSON data structure.
Note that: The string search is case-sensitive and looks for a match within the structure of the JSON data, not just as a substring in the file content. If the directory does not contain any JSON files or if no JSON files contain the string, an empty list is returned.
The function should output with:
list: A list of file paths (str) containing the string within their JSON data."

OUTPUT:
{"0": "Search for a specific string within the JSON data of files in a given directory and its subdirectories. ",
"1": "This function recursively scans the specified directory for JSON files, ",
"2": "then checks each file to see if the given string is present within the JSON data structure.",
"3": "Note that: ",
"4": "The string search is case-sensitive and looks for a match within the structure of the JSON data,",
"5": " not just as a substring in the file content. ",
"6": "If the directory does not contain any JSON files or if no JSON files contain the string, an empty list is returned.",
"7": "The function should output with:",
"8": " list: A list of file paths (str) containing the string within their JSON data."}

Fig. 4. Demonstration of an ICL example used in LLM-based splitter.

As for the *modifier*, we decided to retain the simplicity and rigor of SHAP's removal operation. Therefore, we adopt the previously-mentioned altering function $\mu_r(f_i) = \langle\rangle$ which removes tokens related to a given feature f_i. This choice ensures consistency with the Shapley formulation, where feature importance is defined in terms of the presence or absence of a feature. While removing features may produce inputs that are partially out-of-distribution, this behavior is inherent to perturbation-based explainability methods and is widely adopted in prior work [46].

Finally, the *comparator* depends on the type of artifact produced by the target model M in the task, which is again source code or natural language text. As for the former, we use CodeBLEU [60], a metric designed to assess the similarity of candidate and reference code considering (i) the n-grams overlap through a weighted BLEU [58], (ii) the syntax through Abstract Syntax Tree, and (iii) the semantics via data-flow. For the latter, instead, since the output is natural language text, we adopt BERTScore [80], an effective metric to measure semantic similarity between textual contents through contextual embeddings. Note that we adopt similarity metrics rather than exact-match-based criteria. In code generation, binary signals such as pass/fail are not suitable, as both the original and perturbed outputs may fail, returning identical scores (*e.g.,* 0 vs. 0) and preventing any meaningful comparison. Similarly, exact match is too strict for both tasks: Even semantically equivalent outputs (*e.g.,* two correct implementations or paraphrased summaries) would be treated as completely different. In contrast, CodeBLEU and BERTScore provide continuous similarity scores that capture syntactic and semantic variations that allow a more informative estimation of feature contributions.

`FeatureSHAP` in Action. We first provide a concrete example to help the reader intuitively grasp the capabilities of `FeatureSHAP`. Fig. 5 illustrates a feature attribution scenario for a code generation task. The left portion shows the input prompt, while the right one displays the code generated by the LLM. `FeatureSHAP` evaluates the contribution of each semantically meaningful feature in the input and visualizes their relative influence using varying shades of green – darker shades indicate a higher contribution to the generated output. In this example, the top two features alone account for over ∼78% of the total attribution budget (normalized to one), indicating that a significant portion of the model's behavior can be explained by a small set of semantically rich input segments highlighting the core behavior of the underlying method.

Computational Efficiency. The computational cost of `FeatureSHAP` depends mainly on two factors: (i) the number of features produced by the splitter, and (ii) the Monte Carlo sampling ratio used to approximate Shapley values, which determines the percentage of the total number of combinations of features that will be sampled. Because features are larger semantic units, `FeatureSHAP` usually requires far fewer evaluations than token-level methods such as TokenSHAP or LIME. For example, an input of 200 tokens might be grouped into only 8 features, shrinking the theoretical space from 2^{200} to 2^8. Monte Carlo sampling might then further reduce the number of coalitions (*i.e.,* subsets of features) that must be evaluated when the sampling ratio is lower than 1, making the approach tractable in practice. On the other hand, the comparators (*e.g.,* CodeBLEU or BERTScore) introduce extra overhead, since each perturbed output must be scored with a non-trivial metric. In summary, `FeatureSHAP` theoretically strikes a better balance between explainability and efficiency than token-level attribution, but its runtime remains non-trivial and sensitive to the complexity of some of the design decisions made in terms of splitter, modifier, and (especially, in our specific instantiation) comparator.

Difference with respect to token-level attribution. While `FeatureSHAP` is conceptually inspired by prior Shapley-based methods, it introduces a fundamental shift in how the attribution problem is formulated. Existing approaches work over tokens, treating each token as a player in the Shapley game. In contrast, `FeatureSHAP` defines the Shapley game directly over semantically meaningful features, such as documentation segments or code blocks, which constitute the atomic units of attribution.

This design choice changes the underlying coalition space from token-level combinations to feature-level combinations, resulting in a different attribution problem and search space. Consequently, `FeatureSHAP` is not a

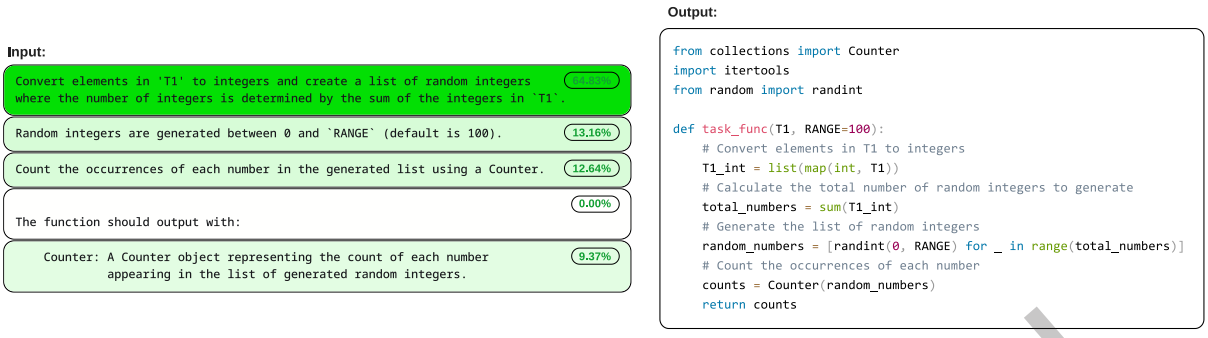

Fig. 5. FeatureSHAP feature attribution for code generation. The darker the highlight, the higher the influence of that feature on the generated code.

simple aggregation of token-level scores, but a reformulation of the attribution problem that aligns the explanation process with domain-specific abstractions used in software engineering.

Finally, FeatureSHAP introduces software-engineering-specific mechanisms that are absent from token-level attribution methods. In particular, it relies on task-aware splitters to identify semantically meaningful features (e.g., natural-language requirements or code blocks extracted through AST-based decomposition) and on task-specific comparators to estimate the contribution of each feature in black-box settings. Together, these design choices customize the Shapley framework for software engineering tasks while preserving the underlying Shapley-value estimation mechanism.

4 Study Design

The *goal* of our study is to evaluate whether FeatureSHAP is both technically effective–accurately determining attribution values for the identified features–and practically acceptable to developers who rely on such explanations. These two dimensions are closely interrelated, since an explainability method must be both reliable and usable to achieve real impact. Consequently, we define the following research questions.

- **RQ_0:** *To what extent does the LLM-based splitter produce stable and human-aligned input features?* With this research question we aim to assess whether the LLM-based splitter produces consistent and meaningful input features.
- **RQ_1:** *To what extent does* FeatureSHAP *assign low attribution to randomly injected non-contributing features?* With this research question we aim to assess to what extent FeatureSHAP can reliably assign low attribution to random noise features that are irrelevant to the model's output.
- **RQ_2:** *To what extent does* FeatureSHAP *assign low attribution to realistic non-contributing features?* With this research question we aim to assess to what extent FeatureSHAP can assign negligible attribution to realistic injected noise that resemble valid features but are designed to be irrelevant.
- **RQ_3:** *To what extent does* FeatureSHAP *facilitate the decision-making process in real-world settings?* This research question aims to evaluate whether the attributions produced by FeatureSHAP are both faithful to the model's decision process and useful to practitioners.
- **RQ_4:** *To what extent is* FeatureSHAP *computationally efficient for practical scenarios?* With this research question, we assess the computational cost of FeatureSHAP by comparing its execution time and monetary cost against *TokenSHAP*.

4.1 Context Selection

The context of our study is composed of *objects* and *subjects*. The *objects* are the LLM splitter, the LLMs and the benchmarks, necessary to answer RQ_0, RQ_1, RQ_2, and RQ_4 while the *subjects* are human participants in a survey that we conduct to answer RQ_3. In the following, we detail the choice of all such elements.

LLM Splitter. As splitter in the code generation task, as defined before, we employ an LLM splitter based on few-shot in-context learning approach using `gpt-4.1-mini`. The objective of such component is to split the input prompt into meaningful features.

LLMs. As explained, `FeatureSHAP` is model-agnostic. Thus, we want to demonstrate its effectiveness on two classes of models: *open-weights* and *closed-weights*. We select LLMs that are representative of the state-of-the-art and widely used in software engineering automation practices. In particular, we focus on `Qwen2.5-Coder` [32] and `GPT-4.1-nano` [54]. The former is a family of code specialized LLMs with state-of-the-art performance on code benchmarks, and it is our representative open-weight model series. We experiment with the three different sizes (3B, 7B, and 32B) to study the effects of the size of the model on the `FeatureSHAP` attribution score. The latter is a lightweight variant of the `GPT-4.1` family. We choose `GPT-4.1-Nano` as it balances the strong performance of GPT-based models on code-specific tasks with a manageable cost and runtime for practical evaluation. For the implementation, we conduct the validation previously described using the models presented in Sec. 4.1. Inference is performed with greedy decoding and a temperature of 0 to foster reproducibility. For the `Qwen2.5-Coder` models, we leverage vLLM [37] to enable efficient inference. We apply Monte Carlo sampling within `FeatureSHAP` using both `Qwen2.5-Coder` and `GPT-4.1-Nano`, setting the sampling ratio to 0, *i.e.,* considering only the essential feature combinations, to reduce computational cost.

Tasks and Benchmarks. We consider two representative and complementary software engineering tasks. The first is *code generation*, which involves synthesizing executable source code from natural language specifications. The second is *code summarization*, which focuses on producing concise natural language descriptions from source code. Together, these tasks capture the two fundamental directions of code–text interaction: generating code from text and generating text from code. To ensure experimental diversity and alignment with established benchmarks in the literature, we rely on input prompts from `BigCodeBench` [82] to evaluate code generation and from `CodeSearchNet` [45] to evaluate code summarization. We select such benchmarks, as they are representative of their task and have become widely adopted in recent literature [4, 13, 20, 22, 31, 32, 68, 75, 78].

`BigCodeBench` [82] is a large-scale Python code generation benchmark designed to assess LLMs in realistic programming scenarios widely used in recent research [2, 32, 38]. It includes 1,140 individual, function-level tasks that span 7 diverse domains and invoke APIs from 139 widely-used Python libraries. We use the `BigCodeBench-Instruct` set in our experiments using the Huggingface's `datasets` package. This benchmark enables us to assess `FeatureSHAP` on function-level tasks typically encountered in real-world software development.

`CodeSearchNet` [33] is a code summarization benchmark modified and prepared by Makharev et al. [48]. We choose this dataset since (i) it is part of `CodeXGLUE` [45], an established widely studied benchmark in the literature [3, 5, 67, 72] and (ii) this version of `CodeSearchNet` further cleaned and processed by Makharev et al. [48], consisting of 846 python `<code,summary>` pairs. This ensures that we can evaluate `FeatureSHAP` on a high quality dataset with a reasonable number of samples.

Note that in this context, we do not need to run test cases (*e.g.,* for code generation) or check the similarity to ground truth (*e.g.,* for code summarization), since the aim of explaining a model's output is independent of the correctness of the generation. However, we still select state-of-the-art benchmarks, as they contain clean and reliable input [48, 82].

Participants. We recruited 37 participants from the academic network of the authors (*convenience sampling*). We only included software engineers, doctoral students, postdoctoral researchers, and early-career faculty – all

of them with solid backgrounds in software engineering or machine learning. Our main selection criteria were two: (i) the participant should be familiar with LLM-based development tools and (ii) they should be capable of critically assessing model explanations. Such criteria ensure that the feedback collected was both informed and reliable.

4.2 Experimental Procedure for RQ_0

The splitter is one of the most important components of `FeatureSHAP`, since the quality of feature-level attribution directly depends on the quality of the extracted features. We therefore first evaluate its reliability. Specifically, in code generation tasks we leverage an LLM to split a prompt into features, as described in Sec. 3. This process is inherently non-deterministic, *i.e.,* it may return different features for the same input prompt. Moreover, the extracted features may not align with those identified by a human. Therefore, to answer RQ_0, we evaluate two dimensions of the LLM splitter: *stability* and *quality*.

Stability. We assess stability by repeatedly running the LLM splitter on the BigCodeBench prompts and measuring the consistency of the produced feature splits. Intuitively, a splitter is stable if, given the same prompt multiple times, it returns the same features most of the time. Conversely, if the output varies significantly across runs, the splitter is unstable.

Therefore, formally, given the LLM splitter σ_n and the set of prompts $P = \{p_0, \dots, p_n\}$, we execute σ_n $R = 10$ times for each prompt p_i, obtaining a set of splits $\{\sigma_n^{(1)}(p_i), \dots, \sigma_n^{(R)}(p_i)\}$. Then, to quantify stability, we look at how often the most frequently produced split appears among the R runs. In other words, for each prompt, we identify the feature decomposition that is returned the highest number of times, and compute its relative frequency.

To quantify this, we define the *stability_ratio* for each prompt p_i as:

$$stability\_ratio(p_i) = \frac{\max_{s \in \mathcal{S}_i} \left| \left\{ r \in \{1, \dots, R\} \mid \sigma_n^{(r)}(p_i) = s \right\} \right|}{R},$$

where $\mathcal{S}_i$ is the set of unique splits observed for p_i. For example, if for a given prompt the LLM splitter produces the same feature split in 8 out of 10 runs, while the remaining 2 runs are different, the stability ratio for such an instance is equal to 0.8. Values closer to 1 indicates higher stability. To obtain a global view, we aggregate $stability\_ratio(p_i)$ across all prompts in P and report summary statistics such as mean, median, and distribution.

Quality. We assess the quality of the LLM-based splitter by measuring how closely its produced features align with a human-defined reference. To this end, we construct a ground-truth dataset by randomly sampling 400 instances from the full set of 1,140 BigCodeBench prompts, corresponding to a statistically significant sample with a 95% confidence level and a margin of error of approximately ±4%. One of the authors manually split each sampled prompt into a sequence of features, which we treat as the reference segmentation.

To enable a principled analysis, we reformulate the splitting task as a text segmentation problem. Specifically, each feature decomposition is interpreted as a segmentation of the input token sequence, where features correspond to contiguous segments and boundaries mark the end of each feature. For example, given a tokenized prompt $\langle t_1, t_2, t_3, t_4, t_5 \rangle$, a feature decomposition such as $\langle \{t_1, t_2\}, \{t_3, t_4\}, \{t_5\} \rangle$ can be represented by placing boundaries after t_2 and t_4. This allows us to represent both predicted and reference features as sequences of boundary indicators over the same token sequence.

We then compare the features produced by the LLM splitter against the human annotations using two standard metrics from the text segmentation literature: *Pk* [11] and *WindowDiff* [59]. Both metrics evaluate the extent to which two segmentations disagree by sliding a fixed-size window of length k over the token sequence and penalizing inconsistencies in segment boundaries. Following standard practice, k is set to half of the average segment length of the reference (human) segmentation for each instance.

In particular, *Pk* measures the probability that two positions at distance k are incorrectly classified as belonging to the same or different segments, while *WindowDiff* extends this by penalizing discrepancies in the number of boundaries within each window. The metrics are computed independently for each prompt and then aggregated across the dataset. Both *Pk* and *WindowDiff* return scores in the range $[0, 1]$, where lower values indicate higher similarity between the predicted and reference segmentations.

As a baseline, we consider a simple *point splitter*, which segments the prompt at sentence boundaries (*i.e.,* on punctuation such as periods). We compare the LLM splitter against this baseline using paired statistical testing. Specifically, we employ the Wilcoxon signed-rank test [77] to assess the significance of differences, and we report effect sizes using Cliff's delta [24]. When performing multiple comparisons, we adjust p-values using Holm's correction [28].

4.3 Experimental Procedure for RQ_1

To answer RQ_1, we develop a noise injection methodology based on Horovicz et al. [29]. The procedure consists of three steps. First, we perform *noise injection* into original prompts from the benchmarks described in Section 4.1. Second, we *filter the samples* and retain only the cases where model output remains unchanged after noise injection, ensuring that the injected tokens are genuinely irrelevant to the generation. Third, we quantify the attribution gap by computing a *Noise Score*, defined as the attribution assigned to injected noise features across the filtered instances. If `FeatureSHAP` works as intended, it should consistently assign an attribution score close to 0 to the irrelevant injected noise. Finally, following Horovicz et al. [29], we compare `FeatureSHAP` against a *random-based attribution* baseline. In addition, we compare `FeatureSHAP` against the *TokenSHAP* baseline [29].

Noise Injection. Consider a prompt i that is being tested among the benchmarks. First, we identify the features of i using the compatible splitter of `FeatureSHAP` based on the type of input (σ_n for natural language and σ_c for source code). We obtain a sequence of features, $\langle f_1, \ldots, f_n \rangle$. Then, we select a noisy feature nf using the strategy we will describe later and inject it at a random position. To ensure that injected noise is comparable to the features within the prompts, we make sure that nf is about the same length as the other features, allowing at most a three-total difference.

In detail, the strategy we adopt in this setting is the *nonsensical* injection. We insert an out-of-distribution sentence selected from a pool of 20 seed sentences generated using GPT. To build this pool, we prompted GPT to produce self-contained statements that are syntactically well-formed but conceptually incoherent. We manually reviewed the pool to ensure that none of them expresses an actionable instruction, a programming task, or any form of meaningful guidance that could plausibly influence the model's behavior. The outcome is a set of linguistically plausible sentences that are irrelevant to the task and therefore serve as genuine noise when injected into a prompt. This setting evaluates whether `FeatureSHAP` correctly assigns negligible attribution to content that cannot influence the behavior of the model by construction. An example of such noise is the feature "*Zebras navigate labyrinthine caves using sonar echoes that humans cannot perceive*" inserted into a sorting function prompt.

Instances Filtering. In noise-injection, to ensure the noise feature is truly irrelevant, we must make sure it does not contribute to the model's output by construction. Therefore, we only conduct the evaluation on cases where noise features are genuinely irrelevant to model output. Our output filtering process retains only instances where injected noise does not alter the behavior of the model. In particular, for each instance of each benchmark, we perform the following steps. First, we generate the baseline output $o = M(i)$ from the original prompt i. Second, we create the noisy version of the prompt (*nonsensical* injection), obtaining the variants ni_{rand}. We then run the model on the variant, obtaining the candidate output $O' = M(ni_{\text{rand}})$. Specifically, we retain the pair $\langle ni_{rand}, M(ni_{rand}) \rangle$ only if $O' = o$, *i.e.,* the injected noise does not change the original output. To this aim, we use exact match to make sure that the injected noise has no impact whatsoever on the output (neither

from a functional nor from a non-functional point of view). All other variants are discarded. To have a tractable computation time for the calculation of exact attribution scores using FeatureSHAP, we filter out instances with more than 12 features since computing the exact Shapley values is computationally expensive (it requires to consider all possible feature coalitions of $2^{|F|}$. With this step, we filtered approximately 13% of the samples from BigCodeBench and 6% from CodeSearchNet.

Metrics. We measure FeatureSHAP's capability to assign low attribution scores to non-contributing features using the *Noise Score* metric, which quantifies, for each instance, the attribution assigned to the injected noise feature. Specifically, given an instance I with an injected noise feature nf, we compute its *Noise Score* as: $S_I = A(nf)$ where $A(nf)$ denotes the attribution score assigned to the noise feature nf. A lower value of S_I indicates that the method successfully minimizes the importance assigned to non-contributing features, while higher values suggest that it incorrectly attributes relevance to them.

We compute the distribution of *Noise Scores* for all evaluated instances $\mathcal{D}$ in terms of mean and median. To statistically compare FeatureSHAP against the baselines (more on this later), we perform pairwise tests on the per-instance *Noise Scores* using the Wilcoxon signed-rank test [77], and Cliff's delta effect size [24]. For multiple comparisons, p-values are adjusted following Holm's correction [28].

Note that we focus our evaluation on noise features only. We do this since those are the only elements for which non-contribution can be established with certainty, given our construction and filtering criteria. In contrast, original features from the prompt may naturally receive low or even zero attribution without letting us be able to determine whether they truly contribute or not.

Baselines. We compare FeatureSHAP with a *random-based attribution baseline*. The latter assigns random attribution scores between 0 and 1 to each feature. Since the sum of attributions should be 1, we normalize such scores dividing them by the sum of the randomly-assigned scores. This serves as a lower-bound performance indicator that establishes the minimum threshold that any meaningful attribution method should exceed.

In addition, we compare FeatureSHAP with the *TokenSHAP* baseline. Since TokenSHAP assigns an attribution score to each input token such that their sum is equal to 1, we aggregate the scores of the tokens forming each feature. Specifically, given a feature $f = \langle t_1, \ldots, t_m \rangle$, we compute its attribution as $A(f) = \sum_{j=1}^{m} \phi_{t_j}$, where ϕ_{t_j} denotes the TokenSHAP attribution score of token t_j. This allows for a fair comparison with FeatureSHAP.

4.4 Experimental Procedure for RQ_2

To answer RQ_2, we develop a cross-sample injection methodology that evaluates FeatureSHAP under more realistic and challenging conditions. While RQ_1 serves as a foundational check, verifying that FeatureSHAP correctly assign low attribution to irrelevant out-of-distribution injected noise features, RQ_2 assesses whether FeatureSHAP can do the same in a more realistic and challenging setting. As for the previous, we first inject a noise feature, then filter the samples that allow model outputs to remain unchanged after noise injection, and finally compute the Noise Score.

Noise Injection. For this RQ, we employ an injection methodology that we refer to as *cross-sample* injection. In this setting, we insert one feature from a *different* sample of the *same benchmark e.g.,* a feature taken from another CodeSearchNet instance when in the code summarization task, or a feature taken from another BigCodeBench input when in the code generation task. For each prompt under test, we first extract its features $\langle f_1, \ldots, f_n \rangle$. We then compute the average feature length *i.e.,* measured as the number of tokens in each feature. This is used to build a pool of candidate features taken from all *other* prompts in the dataset. We retain only those candidates whose length ranges within a three-token window around this average. This ensures that the injected noise is consistent with the form of the original features. From the resulting pool, we randomly select one candidate feature and inject it at a randomly chosen position in the original feature sequence, leaving the rest of the

Prompt for LLM-based Feature Attribution

```
SYSTEM_PROMPT: You are a code feature attribution analyst.
Your task is to evaluate the importance of individual features in contributing to a given model output.
Feature attribution scoring measures how much each feature contributes to the meaning captured in the model output.
A higher score (1) indicates that the feature is more essential for understanding what the code does, while a lower score (0) indicates
the feature is less relevant or even misleading.

CRITICAL REQUIREMENTS:
- All attribution scores MUST be between 0.0 and 1.0
- The sum of all attribution scores MUST equal exactly 1.0
- Provide exactly one score per feature"""

USER_PROMPT:
Prompt: {prompt}
Model output: {model_output}
Please assign an attribution score to each of the following features: {features}
Remember that the sum of the feature attribution scores has to be 1.
```

Fig. 6. Prompt template for the LLM-as-an-attributor baseline. Here, `{benchmark_prompt}` refers to the prompt from the benchmark, `{model_output}` refers to the output of the model based on the `{benchmark_prompt}`, and `{features}` refers to the features obtained from the splitter.

prompt unchanged. This setting evaluates FeatureSHAP's ability to detect contextually inappropriate features, *e.g.,* "*implement binary search*" in a prompt about string manipulation.

Instances Filtering. As in RQ_1, to ensure that the injected feature is irrelevant, we only evaluate cases in which the cross-sample feature does not affect the model's output.

For each prompt i in the benchmark, we first generate the baseline output $o = M(i)$ using the original prompt. We then construct its cross-sample variant ni_{cross} by injecting one feature taken from a different sample of the same benchmark, as described above, and obtain the candidate output $O' = M(ni_{\text{cross}})$. We retain the pair $\langle ni_{\text{cross}}, M(ni_{\text{cross}})\rangle$ only if $O' = o$, *i.e.,* the injected cross-sample feature does not change the original output. Consistently with RQ_1, we use exact string matching between o and O' to ensure that the injected feature has no impact on the model output, neither from a functional nor from a non-functional perspective, while the other variants are discarded.

Metrics. We follow the same principles as those used in RQ_1. Consequently, for each retained instance I with injected cross-sample feature nf, we compute the *Noise Score* as $S_I = A(nf)$ where $A(nf)$ denotes the attribution assigned by FeatureSHAP to the injected feature. The lower values of S_I indicate that the method correctly assigns negligible attribution to the injected feature.

We then analyze the distribution of the noise scores in all filtered instances $\mathcal{D}$ using the mean and median. To compare FeatureSHAP against the baselines, we again performed pairwise Wilcoxon signed-rank tests [77], and computed Cliff's delta effect size [24]. Multiple comparisons are corrected using Holm's correction [28].

Baselines. We compare FeatureSHAP with three baselines. First, as in the previous section, we include a *random-based attribution baseline* and the *TokenSHAP* baseline. The former assigns random values between 0 and 1 to each feature and normalizes them to sum up to 1, while for the latter, that returns an attribution score for each input token, we aggregate such scores for each token that makes a feature. However, in the cross-sample

setting of RQ_2, we complement these baselines with an additional one to better capture this more challenging scenario.

Inspired by LLM-as-a-judge evaluation paradigms [6, 17, 76], which show that LLMs can approximate human judgments in assessing software artifacts, we adopt an *LLM-as-an-attributor* baseline to complement the existing baselines in this more challenging setting. This choice is motivated by the fact that cross-sample noise represents a more realistic and challenging setting. Under these conditions, an *LLM-as-an-attributor* baseline provides a stronger and more realistic comparison, better reflecting a human annotator when judging feature attribution. In detail, for the *LLM-as-an-attributor* baseline, we use `GPT-5-mini` with zero-shot prompting to assign attribution scores. We engineered the prompt in Fig. 6 that instructs the model to behave as a feature attribution analyst and to distribute importance values between the input features. Specifically, the model must assign a real value score between 0 and 1 to each feature, under the constraint that all scores sum to exactly 1.0, thus producing a normalized importance distribution. This setup allows a direct comparison between the distribution generated by the *LLM-as-an-attributor* and the attribution scores computed by `FeatureSHAP`.

4.5 Experimental Procedure for RQ_3

To answer RQ_3 and evaluate the practical effectiveness of `FeatureSHAP`, we conducted a survey with 37 practitioners. Each participant was asked to evaluate six attribution examples generated by `FeatureSHAP` in the two tasks under study: three from code generation and three from code summarization. This practitioner-focused evaluation is essential because assigning low attribution scores to irrelevant features is a necessary but not sufficient condition to conclude that the assigned scores are correct. Indeed, such a preliminary evaluation does not reveal whether the explanations are actually *useful* to humans. Previous work by Li et al. [41] highlights this gap, showing that developers often perceive existing explanation methods as misaligned with their reasoning, raising concerns about practical adoption. Our survey directly addresses this issue by asking practitioners to judge whether `FeatureSHAP`'s feature-level attributions align with their own understanding and provide actionable support for decision-making.

Building on this motivation, we designed the survey to cover a diverse set of examples that would allow participants to judge the clarity and usefulness of the attributions. The central part of the survey consisted of an evaluation of the explanations generated by `FeatureSHAP` on 3 examples of code summarization and 3 code generation. This lead to a total of six examples. Such an amount of examples allows us to gather sufficient data for our analysis and limit participants time and effort needed to run the survey, avoiding either imprecise or lazy answers that could compromise our study results [34, 43]. To select the examples, we used the following procedure. First, we computed descriptive statistics in terms of the size (number of features) of the instances in the benchmarks we considered. Then, for each dataset, we selected three representative samples according to length: one at the first quartile, one at the median, and one at the third quartile of the length distribution. For each selected instance, we run GPT-4.1-nano to perform the task (either code generation or summarization) and used `FeatureSHAP` to generate the explanation by assigning attribution scores to each feature. We chose GPT-4.1-nano because it is one of the selectable models in GitHub Copilot, making it representative of configurations accessible to developers. Note again that we are not interested in task accuracy but rather in the quality of the explanations.

For each example, participants were shown an input-output pair, along with the corresponding `FeatureSHAP` attributions to the input features in percentage form. These attributions were also visualized using color-coded highlights, where the intensity of the color indicated the estimated contribution of each feature to the model's output. Participants were asked to evaluate the *relevance* and *clarity* of such attributions using a combination of Likert-scale ratings [42] and open-ended responses. The evaluation questions were designed following the framework proposed by Marcinkevics et al. [49], which distinguishes between two complementary dimensions: (i) *Faithfulness:* Do the attributions accurately reflect the true influence of each input feature on the model's

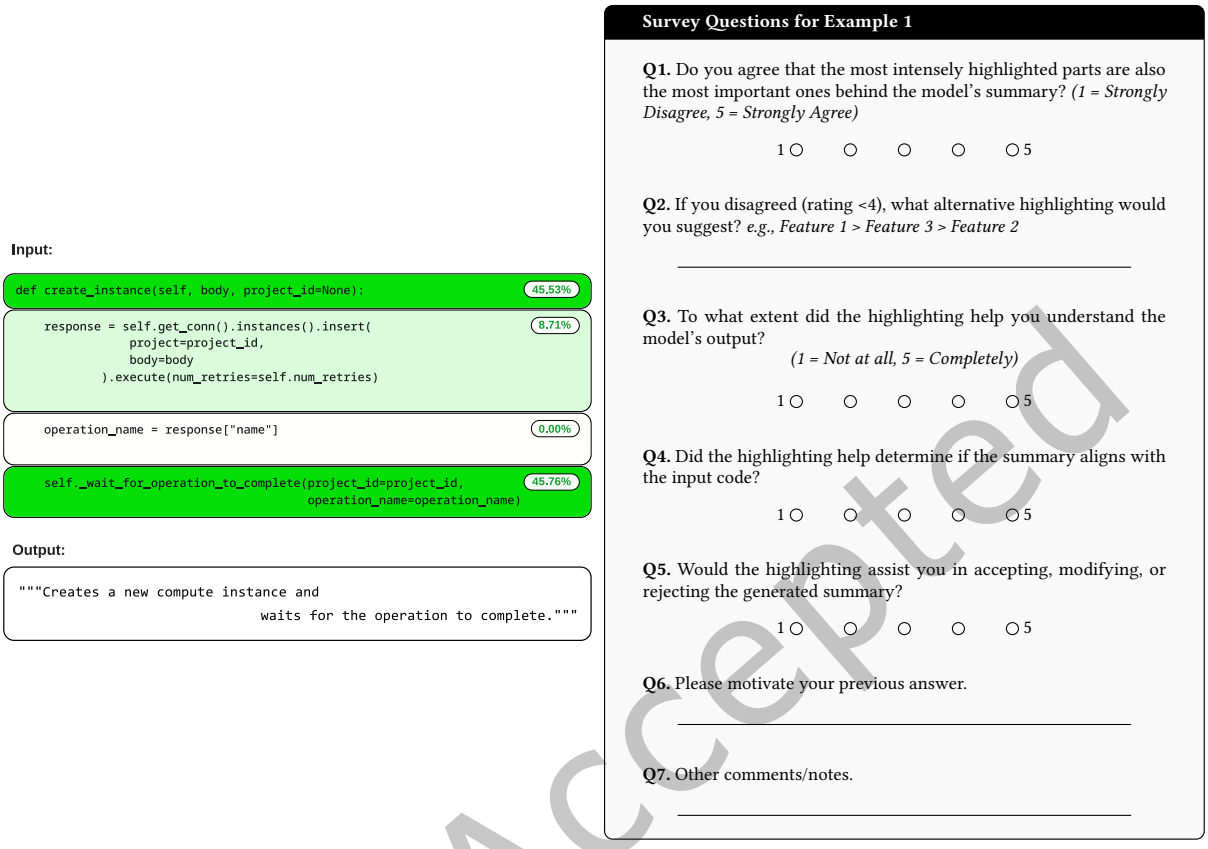

Fig. 7. An example of a code summarization FeatureSHAP attribution from our survey study.

output? We investigated this by asking participants whether the most highly attributed features aligned with their own judgment of importance; (ii) *Usefulness:* Are the attributions understandable, actionable, and supportive of real-world development tasks such as reviewing, accepting, or modifying model outputs? This was evaluated by assessing whether the attributions aligned with the task requirements and whether the participants perceived them as helpful for decision-making.

Each participant reviewed all six examples and provided feedback addressing both faithfulness and usefulness. Fig. 7 illustrates one of the examples shown to participants when evaluating code summarization FeatureSHAP attribution. Each example included both the model's input (*e.g.*, the function to be summarized), the output generated by the LLM, and the corresponding attributions produced by FeatureSHAP, visualized as feature-level highlights. Participants were asked to evaluate the *faithfulness*–captured by **Q1**–(Fig. 7), and *usefulness* that is mapped onto the remaining questions (**Q3-Q5**).

We present both quantitative and qualitative findings derived from participant responses (Sec. 5). Quantitatively, we analyze the distribution of Likert-scale responses across six examples and visualize them using box plots to identify trends in perceived faithfulness and usefulness. We also report qualitative feedback from open-ended responses, highlighting insights from participants found particularly informative, such as alignment with model

behavior and areas for improvement. Together, these findings offer a nuanced view of how FeatureSHAP is perceived in practical SE tasks.

4.6 Experimental Procedure for RQ_4

Explainability techniques must not only provide informative explanations but also incur a reasonable computational cost. Therefore, to answer RQ_4, we analyze the computational cost associated with running FeatureSHAP and compare it against *TokenSHAP*. We selected *TokenSHAP* because it is a token-level perturbation-based explainability technique and therefore provides a suitable reference point to assess the computational implications of operating on features rather than individual tokens. Specifically, we measure and compare both the execution time and the monetary cost required by the two methods.

First, we construct a statistically significant sample by randomly selecting 400 instances from each benchmark, *i.e.,* BigCodeBench and CodeSearchNet, corresponding to a 95% confidence level with a margin of error of approximately ±3.95% and ±3.56% for BigCodeBench and CodeSearchNet, respectively. This allows us to evaluate both approaches on code generation and code summarization tasks. Then, for each benchmark (BigCodeBench and CodeSearchNet), we consider the corresponding sampled instances and, for each of them, we independently execute both FeatureSHAP and the *TokenSHAP* baseline. Each approach is run sequentially on the same input. We repeat this process for each model we considered, *i.e.,* Qwen2.5-Coder-3B, Qwen2.5-Coder-7B, Qwen2.5-Coder-32B, and GPT-4.1-Nano. Experiments are conducted on a machine equipped with an NVIDIA H100 GPU.

For both FeatureSHAP and *TokenSHAP*, we measure the full attribution process end-to-end. This includes all steps required to obtain the final feature- or token-level attribution scores, starting from input processing (*e.g.,* splitting), model inferences, up to the attribution values. For each instance, we record the total execution time required by each method to produce attribution scores. We then analyze the distribution of such times separately for each model and benchmark combination. In addition to execution time, we analyze the monetary cost of both methods in realistic deployment scenarios, particularly when explaining closed-weight models accessed through APIs (*e.g.,* GPT-4.1-Nano). In this setting, the cost depends on both the number of model invocations and the number of processed tokens. For FeatureSHAP, we compute the total attribution cost by accounting for all API-based operations required to generate an explanation, including feature extraction when an LLM-based splitter is employed and the subsequent model invocations required during the perturbation process. For the code summarization task, feature extraction relies on a deterministic parser and therefore does not incur additional API-related costs. For *TokenSHAP*, we compute the total attribution cost as the cost associated with all model invocations performed during the perturbation process. We report the average cost per explained instance and statistically compare the two approaches.

We estimate these costs by tracking, for each instance, (i) the number of API calls performed by each component, and (ii) the total number of input and output tokens processed. We then compute the corresponding monetary cost using the official pricing of the target API model.[4] For open-weight models (*e.g.,* Qwen2.5-Coder), monetary cost is not directly applicable; therefore, we report execution time and the number of model invocations as indicators of computational effort.

We report descriptive statistics of both execution time and monetary cost. To assess whether differences in execution time and monetary costs are statistically significant, we perform pairwise comparisons using the Wilcoxon signed-rank test [77]. We also compute effect sizes using Cliff's delta [24]. Multiple comparisons are corrected using Holm's correction [28].

[4]We use the publicly available pricing of GPT-4.1-Nano at the time of experimentation.

5 Results

In this section, we first report the evaluation of the LLM-based splitter performed in RQ_0, then we show our findings on FeatureSHAP's ability to assign low attribution to injected noise, starting with the preliminary check performed in RQ_1 and followed by the more realistic setting performed in RQ_2. We then present the results of our practitioner-focused user study in RQ_3, examining the faithfulness and usefulness of the explanations produced by FeatureSHAP. Finally, we report the results of the execution time and monetary costs performed in RQ_4.

5.1 RQ_0: LLM Splitter Stability and Quality

Stability. We report the results of the stability analysis of the LLM splitter across the BigCodeBench prompts using the *stability_ratio* metric. The mean stability ratio is ∼0.86, indicating that, on average, the splitter produces the same feature decomposition in approximately 8–9 out of 10 runs for a given prompt. The median is 1.0, showing that for at least half of the prompts the splitter returns identical feature decompositions across all runs. The distribution ranges from a minimum of 0.2 to a maximum of 1.0, showing that while a small number of prompts exhibit low consistency, the majority achieve near-perfect agreement. This is further supported by the standard deviation of 0.192, which indicates moderate variability concentrated in a limited subset of less stable cases. Finally, the results demonstrate that the LLM splitter is stable, reliably producing consistent feature decompositions across repeated runs for most prompts.

Quality. We report the results of the quality evaluation by comparing the feature decompositions produced by the LLM splitter against the human-defined reference using *Pk* and *WindowDiff*. The LLM splitter achieves a mean *Pk* score of 0.121 and a mean *WindowDiff* score of 0.128. Since both metrics measure segmentation disagreement, with lower values indicating better alignment, these results suggest that the splitter produces features that closely match those identified by a human. In practical terms, this means that the LLM splitter is able to split prompts into semantically coherent units that largely reflect how a human would naturally decompose the input. When compared against the point splitter baseline, the LLM splitter consistently achieves lower disagreement scores on both metrics. For example, the LLM splitter obtains 0.121 and 0.128 for *Pk* and *WindowDiff*, respectively, compared to 0.259 and 0.281 for the baseline. All pairwise comparisons are statistically significant ($p < 0.05$ after Holm correction) with large effect sizes ($|\delta| \approx 0.80$–0.83), indicating a consistent improvement over the baseline.

Answer to RQ_0.

The LLM splitter is stable and produces features that closely align with human-defined ones.

5.2 RQ_1: Random Noise Attribution Scores

In Tab. 1 we present our experimental results for RQ_1. These are organized by task (Code Generation and Code Summarization) and models (`Qwen2.5-Coder-{3,7,32}B` and `GPT-4.1-Nano`). We compare FeatureSHAP against a *Random-baseline* and *TokenSHAP* under the *nonsensical* injection setting. Each entry reports: (i) the *Noise Score*: computed per instance as the attribution assigned to the injected noise feature, summarized using the Mean and Median across all instances, (ii) *#Samples*: instances retained after applying filtering criteria from Sec. 4, (iii) the statistical comparison *vs* FeatureSHAP, reported through the adjusted *p-value* from the Wilcoxon signed-rank test and Cliff's δ effect size with its corresponding magnitude *i.e.,* (N)egligible, (S)mall, (M)edium, or (L)arge. We highlight in bold the attributor and the relative *Noise Score* values that outperform each other according to statistical tests ($p < 0.05$). If no statistically significant difference is observed ($p \geq 0.05$), the values are reported in regular font.

On Code Generation. For the code generation task, FeatureSHAP always produces lower *Noise Scores* than the *Random-baseline* and *TokenSHAP*. The medians for FeatureSHAP are 0 across all models, indicating that in at

Table 1. Comparison of FeatureSHAP, *Random-baseline*, and *TokenSHAP* for code generation and code summarization tasks (nonsensical setting). Lower is better.

Task	Model	Attributor	Nonsensical				
			Noise Score		#Samples	vs FeatureSHAP	
			Mean	Median		p-value	δ
Code Generation	QC-3B	**FeatureSHAP**	**0.011**	**0.000**		–	–
		Random	0.168	0.151	245	< 0.05	**-0.9 (L)**
		TokenSHAP	0.044	0.010		< 0.05	**-0.5 (L)**
	QC-7B	**FeatureSHAP**	**0.021**	**0.000**		–	–
		Random	0.161	0.140	395	< 0.05	**-0.9 (L)**
		TokenSHAP	0.047	0.000		< 0.05	**-0.3 (M)**
	QC-32B	**FeatureSHAP**	**0.010**	**0.000**		–	–
		Random	0.167	0.158	341	< 0.05	**-0.9 (L)**
		TokenSHAP	0.054	0.000		< 0.05	**-0.4 (M)**
	GPT-4.1-Nano	**FeatureSHAP**	**0.029**	**0.000**		–	–
		Random	0.180	0.164	127	< 0.05	**-0.8 (L)**
		TokenSHAP	0.102	0.070		< 0.05	**-0.5 (L)**
Code Summarization	QC-3B	**FeatureSHAP**	**0.049**	**0.000**		–	–
		Random	0.173	0.154	159	< 0.05	**-0.7 (L)**
		TokenSHAP	0.070	0.039		< 0.05	**-0.4 (M)**
	QC-7B	**FeatureSHAP**	**0.021**	**0.000**		–	–
		Random	0.194	0.170	284	< 0.05	**-0.9 (L)**
		TokenSHAP	0.054	0.000		< 0.05	**-0.4 (M)**
	QC-32B	**FeatureSHAP**	**0.017**	**0.000**		–	–
		Random	0.194	0.171	203	< 0.05	**-0.9 (L)**
		TokenSHAP	0.041	0.000		< 0.05	**-0.4 (M)**
	GPT-4.1-Nano	**FeatureSHAP**	**0.034**	**0.000**		–	–
		Random	0.181	0.152	217	< 0.05	**-0.8 (L)**
		TokenSHAP	0.097	0.072		< 0.05	**-0.6 (L)**

least half of the instances zero attribution is assigned to the injected noise. FeatureSHAP's means are an order of magnitude smaller than random and consistently lower than *TokenSHAP*. For example, on Qwen2.5-Coder-3B, FeatureSHAP achieves a Noise Score Mean of 0.011 in contrast of the one of the *Random-baseline* of 0.168 and 0.044 for *TokenSHAP*. Similar differences appear on 7B and 32B ($\sim$17× lower on 32B), and on GPT-4.1-Nano ($\sim$6× lower) compared to the *Random-baseline*, while also achieving at least $\sim$2× lower Noise Scores than *TokenSHAP* across all models.

On Code Summarization. For the code summarization task we observe the same patterns. The medians for FeatureSHAP are 0 for all models, and the means are generally 4–11× lower than *Random-baseline* and consistently lower than *TokenSHAP*. For example, on Qwen2.5-Coder-7B, FeatureSHAP produces a Noise Score Mean of 0.021 against the 0.194 of the *Random-baseline* ($\sim$9× lower) and 0.054 of *TokenSHAP* ($\sim$2× lower), and on Qwen2.5-Coder-32B 0.017 vs 0.194 ($\sim$11× lower) of the *Random-baseline*, and vs 0.041 for *TokenSHAP* ($\sim$2× lower).

In both datasets and all models, all pairwise comparisons are statistically significant ($p < 0.05$) with medium and large effect sizes ($|\delta| = 0.4$–0.9), confirming that FeatureSHAP reliably assigns negligible attribution to irrelevant features.

Table 2. Comparison between FeatureSHAP, *Random-baseline*, *TokenSHAP* and *LLM-as-an-attributor* on code generation and code summarization tasks (cross-sample setting). Lower is better.

Task	Model	Attributor	Cross-sample				
			Noise Score		#Samples	vs FeatureSHAP	
			Mean	Median		*p*-value	δ
Code Generation	QC-3B	FeatureSHAP	0.009	0.000	114	–	–
		Random	0.157	0.138		< 0.05	**-0.9 (L)**
		TokenSHAP	0.067	0.025		< 0.05	**-0.6 (L)**
		LLM-Attributor	0.028	0.020		< 0.05	**-0.5 (L)**
	QC-7B	FeatureSHAP	0.018	0.000	210	–	–
		Random	0.170	0.155		< 0.05	**-0.9 (L)**
		TokenSHAP	0.096	0.041		< 0.05	**-0.5 (L)**
		LLM-Attributor	0.022	0.010		< 0.05	**-0.4 (M)**
	QC-32B	FeatureSHAP	0.018	0.000	145	–	–
		Random	0.146	0.127		< 0.05	**-0.9 (L)**
		TokenSHAP	0.098	0.028		< 0.05	**-0.5 (L)**
		LLM-Attributor	0.025	0.010		< 0.05	**-0.4 (M)**
	GPT-4.1-Nano	FeatureSHAP	0.025	0.000	54	–	–
		Random	0.163	0.162		< 0.05	**-0.8 (L)**
		TokenSHAP	0.107	0.066		< 0.05	**-0.5 (L)**
		LLM-Attributor	0.014	0.000		0.868	-0.1 (N)
Code Summarization	QC-3B	FeatureSHAP	0.034	0.000	119	–	–
		Random	0.163	0.151		< 0.05	**-0.8 (L)**
		TokenSHAP	0.060	0.032		< 0.05	**-0.4 (M)**
		LLM-Attributor	0.025	0.020		0.133	-0.4 (M)
	QC-7B	FeatureSHAP	0.019	0.000	186	–	–
		Random	0.180	0.143		< 0.05	**-0.9 (L)**
		TokenSHAP	0.056	0.000		< 0.05	**-0.3 (S)**
		LLM-Attributor	0.026	0.020		< 0.05	**-0.6 (L)**
	QC-32B	FeatureSHAP	0.017	0.000	143	–	–
		Random	0.166	0.148		< 0.05	**-0.9 (L)**
		TokenSHAP	0.049	0.000		< 0.05	**-0.4 (M)**
		LLM-Attributor	0.022	0.020		< 0.05	**-0.6 (L)**
	GPT-4.1-Nano	FeatureSHAP	0.035	0.000	148	–	–
		Random	0.190	0.177		< 0.05	**-0.8 (L)**
		TokenSHAP	0.123	0.088		< 0.05	**-0.6 (L)**
		LLM-Attributor	0.031	0.020		< 0.05	**-0.4 (M)**

Answer to RQ$_1$.

FeatureSHAP consistently outperforms the *Random-baseline* and *TokenSHAP* across tasks and models.

5.3 RQ$_2$: Cross-sample Noise Attribution Scores

In Tab. 2 we shows the results of the comparison of FeatureSHAP with both *Random-baseline* and *LLM-as-an-attributor* as well as *TokenSHAP* on the more realistic setting. The results are organized as in Tab. 1.

On Code Generation. On code generation, FeatureSHAP achieves consistently lower means than both *Random-baseline* and *LLM-as-an-attributor*, and *TokenSHAP* on all Qwen2.5-Coder variants, with statistically significant differences ($p < 0.05$) and medium-to-large effect sizes, *i.e.,* $\delta = -0.8$ to -0.9 against *Random-baseline*, $\delta = -0.5$ to -0.6 against *TokenSHAP*, and $\delta = -0.4$ to -0.5 against *LLM-as-an-attributor*. Medians for FeatureSHAP are 0 across all models, while *LLM-as-an-attributor* shows non-zero medians (0.01–0.02), *TokenSHAP* between 0.025-0.066, and *Random-baseline* between 0.127-0.177. This shows that FeatureSHAP generally assigns

no attribution to the injected noise on at least half of the instances. For `GPT-4.1-Nano`, *Random-baseline* produces higher mean and median (0.163, and 0.162). In the same setting, *TokenSHAP* achieves higher Noise Scores (0.107 mean, 0.066 median), confirming that `FeatureSHAP` assigns lower attribution to noise even when compared to token-level explanations. The *LLM-as-an-attributor* has a slightly lower mean (0.014 vs 0.025) than `FeatureSHAP` and both medians are 0; the difference is not significant ($p = 0.868$, $\delta = -0.1$, negligible). However, looking at the left side of Fig. 8, we can observe that `FeatureSHAP` distribution is closely concentrated around zero compared to the *LLM-as-an-attributor*, which exhibits a greater variability.

On Code Summarization. On code summarization, results are mixed but generally in favor of `FeatureSHAP`. Generally, `FeatureSHAP` consistently assigns lower attributions to noise with respect to the *Random-baseline* and *TokenSHAP* with statistically significant differences and small-to-large effect size ($\delta = -0.8$ to -0.9 against the *Random-baseline* and $\delta = -0.3$ to -0.6 against *TokenSHAP*). Furthermore, for `Qwen2.5-Coder-7B/32B`, `FeatureSHAP` produces lower means than the *LLM-as-an-attributor* (0.019/0.017 vs 0.026/0.022) with statistically significant differences ($p < 0.05$) and large effects ($\delta = -0.6$). In these settings, `FeatureSHAP` also consistently achieves lower Noise Scores than *TokenSHAP* (*e.g.,* 0.019 vs 0.056 and 0.017 vs 0.049), corresponding to improvements of approximately 2–3×.

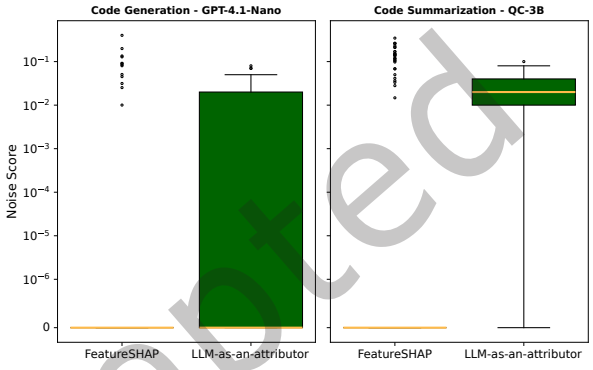

Fig. 8. On the left, *Noise Score* distributions of `FeatureSHAP` and *LLM-as-an-attributor* on `GPT-4.1-Nano` for `BigCodeBench`. On the right, *Noise Score* distributions of `FeatureSHAP` and *LLM-as-an-attributor* on `Qwen2.5-Coder-3B-Instruct` for `CodeSearchNet`. Lower is better.

On the other hand, for `Qwen2.5-Coder-3B`, the mean difference (0.034 vs 0.025) is not significant ($p = 0.133$), while the negative δ shows a tendency towards lower per-instance scores for `FeatureSHAP` despite a few larger outliers increasing its mean. This pattern is visible in the right panel of Fig. 8. `FeatureSHAP`'s median are set to 0 with a compact interquartile range, while *LLM-as-an-attributor* shows a higher median and broader spread. For `GPT-4.1-Nano`, the mean for `FeatureSHAP` is slightly higher (0.035 vs 0.031), yet the Wilcoxon test favors `FeatureSHAP` ($p < 0.05$, $\delta = -0.4$, medium), which, together with the medians (0.000 vs 0.020), indicates that `FeatureSHAP` more often assigns near-zero attribution but occasionally produces higher attributions. Even in this case, *TokenSHAP* produces higher Noise Scores (0.123 mean, 0.088 median), further confirming the advantage of feature-level attribution over token-level methods.

Overall, across datasets and models, `FeatureSHAP` always outperforms the *Random-baseline* and *TokenSHAP*, and either matches or surpasses *LLM-as-an-attributor*. When differences are significant, they favor `FeatureSHAP` with small to large effects. When not significant, distributions still show `FeatureSHAP` concentrated at zero (median = 0), whereas *LLM-as-an-attributor* more frequently assigns non-negligible attribution to noise (*e.g.,* medians of 0.01–0.02).

Importantly, however, `FeatureSHAP` offers greater transparency than *LLM-as-an-attributor*. `FeatureSHAP` SHAP-based attributions process is fully visible to users, instead, the *LLM-as-an-attributor* works as a fully black-box system providing only final scores. In addition, the *LLM-as-an-attributor* inherently leverages its own understanding of text (*i.e.,* natural language and code) to inspect the *content* of each feature. This allows it to detect injected noise directly from its semantic incongruity with the surrounding prompt, rather than from a principled assessment of the model's actual dependency on a feature. In contrast, `FeatureSHAP` computes

the feature attributions strictly from systematic perturbations and model responses, ensuring that attributions genuinely reflect the model's own behavior.

Answer to RQ_2.

FeatureSHAP outperforms the *Random-baseline* and *TokenSHAP*, and performs on par with or better than the *LLM-as-an-attributor*.

5.4 RQ_3: FeatureSHAP Alignment with Human Perception.

In Fig. 9 we provide an overview of the background of the survey participants, in which we show the proportions of participants belonging to five professional categories: Ph.D. students, master's students, industry professionals, assistant professors, and postdoctoral researchers. Most of them (48.6%) are Ph.D. students, while the 21.6% of the participants involved are Master students, and 18.9% Industry Professionals (*e.g.,* software engineers, data scientist, *etc.*).

We report in Fig. 10 the distribution of Likert-scale ratings for responses related to *faithfulness* and *usefulness*, as provided by participants in both code generation and summarization examples of the survey. The left panel summarizes the overall ratings for both code generation (CG) and code summarization (CS) tasks, aggregating all evaluated examples per task. The right panel shows the detailed distributions for each individual example. The upper row corresponds to code generation examples, and the bottom row reports results for code summarization ones.

Quantitative Analysis. Participant ratings indicate that FeatureSHAP produces explanations perceived as both *faithful* and *useful* across the evaluated tasks. As shown in Fig. 10, the distributions of Likert-scale responses for both dimensions tend consistently toward higher values for both tasks, with medians equal to 4 on the 5-point scale. The interquartile ranges span mostly between 3 and 5, showing moderate consensus among participants. This represent that feature-level attributions generated by FeatureSHAP accurately reflect the model's decision process and are practically helpful, according to the participants. The overall boxplots (left panel in Fig. 10) show that ratings for code generation (CG) and code summarization (CS) are similar, suggesting that FeatureSHAP generalizes well across distinct SE tasks involving both natural and programming languages. However, while the distribution for code generation shows slightly wider variability, particularly for *usefulness* with respect to the code summarization counterpart, the median and upper quartiles remain high, confirming that explanations remain consistently well aligned with participants reasoning despite the greater inherent complexity of code generation compared to code summarization.

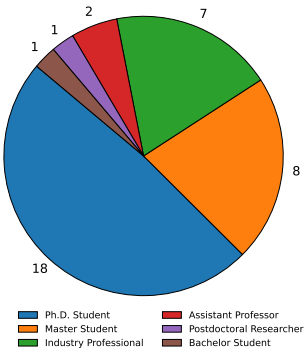

Fig. 9. Participants job positions.

Analyzing the ratings at individual example level (right part of Fig. 10), we observe a similar pattern across all six examples. In detail, for the first code generation example, participants provided uniformly high ratings for both *faithfulness* and *usefulness*, although a more variability is noticed for *usefulness* related ratings. Instead, in the second code generation example, the median rating for *faithfulness* is equal to 3 showing moderate participants alignment with FeatureSHAP attribution scores. For the third code generation example, both *faithfulness* and *usefulness* receives higher ratings, with medians at 4 and upper quartiles extending to 5. The limited variance observed reports that participants consistently perceived FeatureSHAP's explanations as indicative of the factors influencing the model's generation behavior, and as helpful to be leveraged in practice. Moving to the code summarization task, the first example shows that both *faithfulness* and *usefulness* ratings centered around 4 with

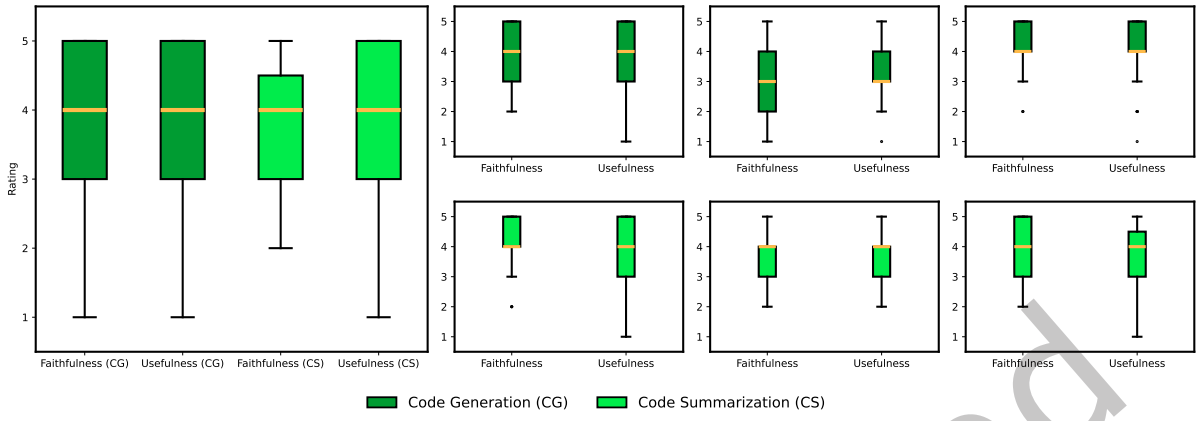

Fig. 10. Participant ratings of FeatureSHAP for faithfulness and usefulness across code generation (CG) and code summarization (CS). The left panel provides an overall summary per task. The right panels show detailed results for individual examples, top row for code generation and bottom row for code summarization.

strong upper quartile for *faithfulness* ratings. In the second summarization example, a slightly wider variability appears in the *faithfulness* ratings, with a median value equal to 4. Finally, for the third summarization example, both *faithfulness* and *usefulness* achieve a median equal to 4, with upper quartiles at 5 and a few low outliers.

In general, for both code generation and code summarization, *faithfulness* median ratings often equal to 4 with the upper quartiles that reaches 5 for the first and third example, while for the second example the median is equal to 3. This consistency indicates that participants largely agreed that the most important features according to FeatureSHAP indeed correspond to the parts of the input that mostly influenced the model to infer the output. *Usefulness* ratings follow a similar trend, with median values equal to 4 for all the examples except for the second code generation and code summarization examples. Importantly, no significant difference emerges between *faithfulness* and *usefulness* ratings within the same example, implying that participants found faithful explanations also directly useful for their assessment of model behavior.

Qualitative Analysis. Looking at the qualitative feedback reported by participants, we observe consistent evidence that FeatureSHAP provides both *faithful* and *useful* explanations across the evaluated tasks. In the code generation examples, participants emphasized the usefulness of the highlighting distribution (*i.e.,* feature attributions) as guidance to understand the model behavior and verify requirement coverage. For example, participants stated *"The highlighting makes it super easy to see what kind of code I'm trying to generate and what the AI focused on"*, *"Highlighting helps to focus on the important parts that are integral for the code"*, or *"The highlighting clearly maps requirements to code sections ..."*. This successfully aligns with our conjecture *i.e.,* splitting in semantic features rather than in single tokens aligns better with practitioners point of view. Others confirmed this aspect more explicitly, noting that *"Highlighting the input according to its degree of influence is useful for focusing attention on the most relevant parts of the request"*. These comments show how participants perceived FeatureSHAP useful, providing transparency for what the model prioritized when producing the output. A minority of participants pointed out limitations, mainly related to imbalances in the attribution scores. Some reported that *"Highlighting is helpful but sometimes misses key requirements ..."*. This kind of comment reveals occasional mismatches between human perception and FeatureSHAP attributions. Instead, comments as the following, *"... I might mistakenly accept generated code that does the highlighting things well and ignore the non-highlighted things ..."*, show that while participants appreciate feature attributions, they manually must cross-validate the code to ensure completeness. In

fact, our goal is not to encourage developers to automatically accept generated outputs based on the explanations, but rather to *support* their reasoning process to promote transparency and awareness of the model generation process. For example, this can enable practitioners to spot unreliable attributions and to critically assess when to trust the model or when to refine the prompt or the code itself. In this sense, explanations serve as decision aids, not as decision replacements. Some of the participants indeed align with our assumption stating that *"... seeing that the model puts more emphasis in the initialization and the middle function rather than the regex code will make me re-prompt it better"*. Nevertheless, the general trend remains positive, with several participants stating that *"It indeed helps to understand whether the model tackled all the requirements in the prompt"*.

Similar patterns emerge for the code summarization task. Participants highlighted the benefit of traceability between the source code and the generated documentation, describing that *"It helps me quickly identify the parts of the code that are considered by the model, and in this specific case that the parts that are given the greatest consideration are precisely the important ones."* and *"This is a very obvious example to see precisely where the model drew its output from. The two parts of the output directly correspond to the two most highlighted parts of the code"*. Such comments suggest that participants found the explanations faithful, and helpful for verifying the consistency between code and the generated summaries. Some participant disagreed, mentioning that *"... the highlighting does not accurately reflect the importance of each feature in the summary"*, indicating that small inconsistencies in attributions intensity could sometimes reduce perceived faithfulness.

In addition, some also suggested newer perspectives and future directions for more human-aligned explanations. For example, the following comment, *"It would be even easier if the text, or the individual feature, were associated with the piece of code"*, suggests for a specialization foreseeing a mapping between feature attributions and output features. One other participant also wondered about a multiple-steps explanation process in which first a more general overview is provided, followed by the possibility to progressively explore finer-grained details of the attribution, *i.e., "... it might be interesting to see the LLM work with more granular information ..."*.

In summary, the qualitative feedback confirms the quantitative findings. Participants perceived `FeatureSHAP` as both *faithful* and *useful*, improving the understanding of the model generation process and supporting decision-making in results inspection. Even when misalignment emerged, `FeatureSHAP` attributions enhanced transparency rather than undermining trust, as participants used such discrepancies to critically assess where the model focus diverged from their own expectations.

Answer to RQ_3.

Participants rate `FeatureSHAP`'s explanations as both faithful and useful across both code generation and code summarization tasks. Qualitative feedback further shows that feature-level attributions help users understand and check the model's behavior.

5.5 RQ_4: Computational Cost of `FeatureSHAP`.

In Fig. 11, we report an overview of the execution time and the monetary costs, comparing `FeatureSHAP` and *TokenSHAP*. All pairwise comparisons discussed in this section are statistically significant after Holm's correction and are associated with large effect sizes; detailed statistics are available in the replication package [1].

Execution Time. Across all considered models and tasks, `FeatureSHAP` consistently requires less time than *TokenSHAP*. This difference is particularly evident in the code generation task, where `FeatureSHAP` completes the attribution process in approximately 15 seconds when using `GPT-4.1-nano` and 14 seconds when using `Qwen2.5-Coder-32B`, compared to approximately 51 and 58 seconds required by *TokenSHAP*, respectively. Similar trends are observed in the code summarization task, where the average execution time of `FeatureSHAP` ranges between 1.5 and 3.4 seconds across all evaluated models, while *TokenSHAP* requires between 9 and 28 seconds.

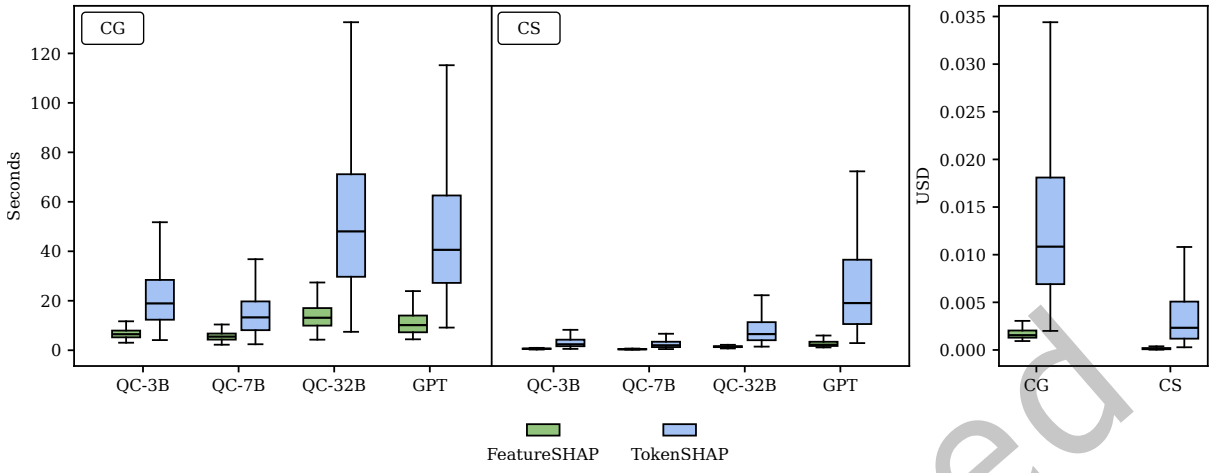

Fig. 11. Comparison of FeatureSHAP and *TokenSHAP* in terms of execution time and monetary cost across code generation (CG) and code summarization (CS) tasks. Lower values indicate better efficiency.

These results show that the computational advantages of FeatureSHAP are systematic rather than model- or dataset-specific.

Model Invocations. To provide additional insight into the observed runtime differences, we report the number of perturbation evaluations performed by each approach. In the code generation task, FeatureSHAP evaluates on average seven feature coalitions per instance, whereas *TokenSHAP* requires approximately 142 token-level perturbations. A similar pattern emerges in the code summarization task, where FeatureSHAP evaluates approximately five coalitions compared to more than 120 perturbations required by *TokenSHAP*. Overall, FeatureSHAP reduces the number of evaluations by more than one order of magnitude across both benchmarks. This reduction indicates that feature attributions can be estimated with substantially fewer model invocations, providing a practical explanation for the execution-time improvements observed in Fig. 11.

Monetary Costs. Since modern LLMs are typically accessed through usage-based APIs, reducing the number of perturbation evaluations has a direct economic impact. When using `GPT-4.1-nano`, FeatureSHAP consistently incurs lower costs than *TokenSHAP* across both tasks. In the code generation task, the average attribution cost decreases from approximately $0.015 per instance for *TokenSHAP* to approximately $0.002 for FeatureSHAP, corresponding to an 8× reduction. The reduction is even more pronounced in the code summarization task, where the average cost drops from approximately $0.005 to less than $0.0002, corresponding to more than one order of magnitude cost reduction. These findings suggest that the practical deployment of feature-level explainability becomes substantially more affordable, particularly when explanations must be generated repeatedly during development activities.

The results provide strong empirical evidence that FeatureSHAP requires substantially lower computational costs than *TokenSHAP*. Across all evaluated models and benchmarks, FeatureSHAP consistently achieves lower execution times and lower monetary costs. Combined with the substantial reduction in the number of perturbation evaluations, these results show that feature-level explainability can be achieved with significantly lower computational requirements than token-level alternatives.

Answer to RQ$_4$.

FeatureSHAP incurs lower computational costs than *TokenSHAP*. Across all evaluated models and tasks, it results in consistently lower execution times and monetary costs.

6 Implications of Our Findings and Future Work

Our study presents strong evidence that FeatureSHAP offers a practical, scalable, and domain-aligned solution to the challenge of explainability in software engineering automation. Based on our findings, we define the following implications for researchers, practitioners, and tool builders.

Researchers. Our findings emphasize the importance of domain-adapted explainability techniques that go beyond token-level reasoning by leveraging semantically-meaningful features that reflect the way developers naturally interpret code.

FeatureSHAP opens new avenues for exploring feature-based attribution in a wider range of tasks, including code-to-code scenarios such as automated program repair, refactoring, and code translation. Indeed, the modular design of FeatureSHAP also allows researchers to explore and specialize the different components (*i.e.,* splitter, modifier, and comparator) for a specific task at hand. For example, regarding the automated program repair (APR) task, the splitter can be adapted to isolate the buggy statement or its corresponding AST subtree as a dedicated feature, while grouping the remaining statements into features of comparable granularity to preserve the structural balance of the code. The modifier can then replace each feature with semantics-preserving alternatives, such as substituting the buggy statement with a minimal repair candidate or a paraphrased equivalent, thus maintaining syntactic validity while probing the model's dependence on that section. Finally, the comparator can leverage code-aware similarity metrics (*e.g.,* CodeBLEU) to quantify how each substitution alters the patch produced by the model. While evaluating such adaptations is outside the scope of the present work, our results suggest that feature-level attribution provides a promising foundation for investigating LLM behavior in software engineering tasks. In general, while initial feature splitting rules may require domain expertise, we have shown that this step can be effectively automated using LLMs or other techniques. This demonstrates that the framework is not only extensible but also scalable, enabling its application in diverse tasks where LLMs explainability is essential. Moreover, our stability analysis suggests that LLM-based feature decomposition can generate consistent feature representations across executions while remaining aligned with human-defined segmentations, supporting its use as a reliable building block for feature-level explainability.

In addition, FeatureSHAP also lends itself to *systematic* analysis of LLMs behavior. Since attributions are computed in a model-agnostic and fully automated way, one can run FeatureSHAP over large batches of generations (*e.g.,* across prompts) to spot recurring failures, spurious correlations, and contrast attribution before and after fine-tuning. For example, given a large dataset of prompts that share a common structure, researchers can identify recurring prompt components (*e.g.,* initial instructions, requirements, constraints, or non-functional specifications) and apply FeatureSHAP to obtain feature-level attributions for each instance. Since the features would correspond to known prompt sections, aggregating these for each instance can show which parts of the prompt the model consistently relies on, enabling the analysis and exploration of the model global behavior. Similarly, running FeatureSHAP on pairs of models, such as a model tuned to instruction and its fine-tuned counterpart, on the same dataset can allow researchers to directly contrast their attributions on the same inputs. Such comparisons can let one study how fine-tuning shifts the model's dependence on specific features, studying why behavior changes and which modifications could have altered model reasoning.

Furthermore, it could be meaningful to connect FeatureSHAP to the mechanistic interpretability [12] through *causal interventions*. In practice, this means testing whether removing or altering specific input features (input-level ablations) or modifying their internal activations (activation patching) leads to predictable changes in

the model's output. Such tests act as a causal sanity check that high-attribution features truly *matter* for the model's behavior. At the same time, it could extend FeatureSHAP with *dependency-aware attribution*, which explicitly accounts for the fact that code features are often interdependent. By estimating interaction effects and using AST-aware masking to preserve syntactic and semantic validity, this enhancement can reveal whether the importance of a feature (*e.g.,* a docstring) comes from its own contribution or from how it works together with related elements like a function signature. These are part of our future agenda.

Practitioners. One immediate benefit of adopting FeatureSHAP from the practitioners' point of view lies in its ability to provide a streamlined explanation that shows which parts of a prompt (or code parts) most strongly influenced the model's output. This, enables developers to better understand, validate, and refine LLM-generated content. As confirmed by our user study, where participants reported improved understanding and decision support, such targeted attribution facilitates the extraction of actionable insights, thereby supporting informed decision-making throughout different stages of the software development life-cycle. In practice, FeatureSHAP can support an iterative development workflow. Consider a developer using an LLM to implement a file-search utility with a prompt containing multiple requirements, such as supporting case-insensitive search, handling large files efficiently, and returning results sorted by relevance. After generating the code, the developer applies FeatureSHAP to inspect which requirements most influenced the output. Suppose the attribution reveals that the relevance-sorting requirement receives high importance while the case-insensitive constraint receives little importance. The developer can then revise and emphasize the neglected requirement, regenerate the solution, and use FeatureSHAP again to verify whether the model now relies on the intended feature. This allows an evidence-based feedback loop that helps developers systematically understand, validate, and refine LLM-generated artifacts.

Another implication pertains to prompt optimization and emerges directly from our empirical observations. Since FeatureSHAP provides fine-grained attribution scores for each semantic input feature, it enables precise refinement of the components that most significantly influence the model's behavior. For example, in a code generation setting with a prompt composed of five distinct features, FeatureSHAP might reveal that just two of them (*e.g.,* f_1 and f_3) contribute over 95% of the total attribution. If these dominant features are found to be poorly phrased or overly verbose, developers can revise them to improve clarity and reduce noise. This targeted revision is valuable in light of recent findings showing that verbose prompts can dilute critical signals and impair model performance [40].

Importantly, this workflow remains computationally affordable in realistic settings. Our efficiency analysis shows that FeatureSHAP generates explanations in a matter of seconds for code summarization and ∼10 seconds for code generation, compared to up to one minute required by *TokenSHAP*. These reductions also translate into lower monetary costs, with attribution costs decreasing by approximately 8× in code generation and by more than one order of magnitude in code summarization. As a result, developers can obtain feature-level explanations quickly and at a substantially lower cost than token-level alternatives.

Beyond performance gains, selective refinement offers important benefits for the *sustainability* of LLM-based workflows. By systematically identifying and eliminating irrelevant or low-impact features, prompts can be compressed significantly while preserving task accuracy. This compression reduces both the cognitive burden and computational overhead of prompt engineering, yielding concrete savings in inference time and API costs, particularly when deployed at scale.

Finally, practitioners can define features that match their own conventions obtaining explanations tailored to singular developer preferences. For example, a developer could manually split a prompt into their own manually-defined features by enforcing the conceptual structure they apply when reasoning about a task. One developer might prefer a more general aggregation such as treating all initial assumptions as "setup logic", the main requirements as "core logic", and all special conditions as "edge-case rules". Another developer might

instead prefer a more fine-grained view, further separating preconditions from parameter constraints, or splitting edge-case handling into individual sub-rules. Over time, such manually crafted features can be leveraged to train a personalized splitter, enabling FeatureSHAP to produce explanations more aligned with that singular developer's reasoning style.

Tool Builders. For tool builders, FeatureSHAP presents a practical opportunity to integrate explainability into development workflows. Our experiments show that competitive attribution quality can be achieved even with low sampling ratio, enabling efficient real-time explanations. This suggests a straightforward IDE integration pattern: default to low-cost, low-sampling-ratio attributions for instant feedback during prompt development. This approach makes feature-level explainability computationally feasible as an always-available development aid rather than a resource-intensive post-hoc analysis, allowing developers to continuously monitor and refine feature contributions throughout the prompt engineering process. To illustrate this vision, an early UI mockup of a potential IDE extension integrating FeatureSHAP is provided in the replication package [1]. Our efficiency results further support this vision, showing that feature-level explanations can be generated substantially faster and at lower cost than token-level alternatives, making their integration into development tools and automated software engineering workflows more realistic. Moreover, the stability of the automated splitter helps ensure that explanations remain consistent across executions.

7 Threats to Validity

Threats to *construct validity* concern the extent to which our experimental setup accurately captures the theoretical constructs under investigation–in this case, whether FeatureSHAP meaningfully explains the behavior of LLMs in software engineering tasks, particularly code generation and summarization. FeatureSHAP, as a method grounded in SHAP, produces explanations that are inherently *local* and instance-specific because Shapley values are computed relative to the set of co-occurring features in a given instance. Contextual variations–such as changes in prompt structure, feature order, or the inclusion of additional semantic elements—can shift the resulting importance distribution. A key threat lies in how we define semantic input features. For code generation, we group tokens into components such as the docstring summary, metadata sections, and function signatures. For code summarization, we segment the code into higher-level blocks (*e.g.,* AST nodes like function definitions and their statements). Although these abstractions are grounded in developer practice, language models may instead rely on statistical cues, subword artifacts, or positional encodings that do not align with our chosen boundaries. Thus, there is a risk that our abstractions capture only correlations rather than the mechanisms that actually drive the model predictions. Furthermore, different feature decompositions may lead to different attribution outcomes, as the definition of the feature space directly determines the Shapley game. Future work should investigate whether feature definitions genuinely reflect what models rely on or merely approximate them. A further threat stems from the perturbation strategy adopted by FeatureSHAP. In line with SHAP-based methods, we rely on feature removal to estimate contributions. However, removing entire semantic units may produce inputs that are partially out-of-distribution, potentially affecting the model's behavior in ways that do not reflect realistic usage scenarios. Although this limitation is inherent to perturbation-based explainability techniques and widely acknowledged in prior work [46], it may influence the stability of the resulting attributions.

A second threat arises from our choice of comparator metrics. In code generation, we use CodeBLEU, while for summarization we rely on BERTScore. Although these are state-of-the-art similarity metrics, alternative continuous comparators may yield different attribution distributions. Assessing this sensitivity would require running the full attribution pipeline for each comparator, resulting in substantial computational cost. For instance, simpler criteria such as pass/fail or exact match are not suitable in our setting, as they often provide limited discriminative power (*e.g.,* identical scores for original and perturbed outputs), preventing meaningful comparison. While our choice is motivated by the need for continuous similarity measures in the Shapley framework, it may

still influence the resulting attributions. Finally, the granularity of features may influence attribution scores. Coarser features may overlook fine-grained contributions, while excessive fine-grained features may reintroduce the complexity and the instability of token-level approaches. Although our design aims to balance interpretability and tractability, identifying the optimal granularity remains an open challenge.

Threats to *internal validity* pertain to methodological choices that may have affected our results. In our study, attribution variance stemming from the inherent non-determinism of LLMs is only partially mitigated, as we relied on greedy decoding for all conditions to keep the experimental procedure tractable.

Threats to *external validity* concern the generalizability of our findings beyond the specific setting of our experiments. First, our analysis is grounded in a specific benchmark of paired prompts derived from the `BigCodeBench` and `CodeSearchNet` benchmarks, which, although widely adopted, may not represent the full spectrum of real-world code generation or summarization tasks. Second, we focus on a single programming language (*i.e.,* Python), and additional work is needed to extend our approach to other languages, such as Java, C++, *etc.* Third, we evaluated two representative model families `Qwen2.5-Coder` (3B/7B/32B) and `GPT-4.1-nano`, spanning open-weights and closed-source settings; although these are widely used and representative of the state-of-the-art, extending the analysis to other models would strengthen external validity.

Threats to *conclusion validity* refer to the correctness of inferences drawn from our experimental data. We report descriptive comparisons across models, datasets, and baselines, but we do not establish causal relationships beyond noise-injection validation. However, to mitigate these threats, we employed statistical procedures (Wilcoxon signed-rank test, Cliff's δ) to compare attribution distributions, and Holm's correction for multiple comparisons. Our assessments also rely on the perception of the practitioner rather than ground-truth explanations of the behavior of the model. Similarly, the human-defined segmentations used as reference in the LLM-based splitter analysis represent one plausible decomposition of the input, and alternative annotators may identify different yet equally valid feature boundaries. Moreover, multi-collinearity among correlated code features (*e.g.,* docstrings, signatures, and implementations) can distort Shapley attributions by artificially distributing importance across dependent elements. Although `FeatureSHAP` provides interpretable results, we cannot claim that these align with the actual internal decision-making processes of the LLMs [14].

8 Conclusions

In this work, we introduced `FeatureSHAP`, the first fully automated black-box explainability framework explicitly designed for software engineering tasks, particularly generative bi-modal tasks, including code generation and code summarization. By attributing model outputs to semantically meaningful features, `FeatureSHAP` narrows the gap between opaque model behavior and the way developers naturally reason about code. Our empirical evaluation shows that `FeatureSHAP` not only outperforms random baselines, LLM-as-an-attributor, and token-level explainability techniques to identify irrelevant inputs, but also produces explanations that align closely with the intuition of practitioners at a lower computational cost. These findings position `FeatureSHAP` at the forefront of explainable software engineering automation, laying the groundwork for future efforts in the area that should focus and examine the statistical rigor of attribution methods, incorporate broader evaluation metrics beyond similarity-based scores, and address the challenges of aligning explanations with human reasoning and expectations.

9 Acknowledgments

This publication is part of the project PNRR-NGEU which has received funding from the MUR – DM 118/2023. This work has been partially supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "DevProDev: Profiling Software Developers for Developer-Centered Recommender Systems", grant n. 2022S49T4W, CUP: H53D23003610001. This work has been partially

supported by the European Union-NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-Based Systems," Grant no. 2022B3BP5S, CUP: H53D23003510006.

References

[1] 2025. Replication Package. https://doi.org/10.5281/zenodo.17950055.
[2] Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, Congcong Chen, et al. 2025. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. *arXiv preprint arXiv:2503.01743* (2025).
[3] Saima Afrin, Joseph Call, Khai-Nguyen Nguyen, Oscar Chaparro, and Antonio Mastropaolo. 2025. Resource-Efficient & Effective Code Summarization. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 224–235.
[4] Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. 2025. OpenCodeInstruct: A Large-scale Instruction Tuning Dataset for Code LLMs. *arXiv preprint arXiv:2504.04030* (2025).
[5] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*. 1–5.
[6] Toufique Ahmed, Premkumar Devanbu, Christoph Treude, and Michael Pradel. 2025. Can llms replace manual annotation of software engineering artifacts?. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 526–538.
[7] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13.
[8] Saranya Alagarsamy, Chakkrit Tantithamthavorn, Wannita Takerngsaksiri, Chetan Arora, and Aldeida Aleti. 2025. Enhancing large language models for text-to-testcase generation. *Journal of Systems and Software* (2025), 112531.
[9] Kenza Amara, Rita Sevastjanova, and Mennatallah El-Assady. 2024. Syntaxshap: Syntax-aware explainability method for text generation. *arXiv preprint arXiv:2402.09259* (2024).
[10] Lakshit Arora, Sanjay Surendranath Girija, Shashank Kapoor, Aman Raj, Dipen Pradhan, and Ankit Shetgaonkar. 2025. Explainable Artificial Intelligence Techniques for Software Development Lifecycle: A phase-specific survey. https://ieeexplore.ieee.org/document/11126697/
[11] Doug Beeferman, Adam Berger, and John Lafferty. 1999. Statistical models for text segmentation. *Machine learning* 34, 1 (1999), 177–210.
[12] Leonard Bereska and Efstratios Gavves. 2024. Mechanistic Interpretability for AI Safety - A Review. *Transactions on Machine Learning Research* (Aug 2024). https://openreview.net/forum?id=ePUVetPKu6
[13] Liuwen Cao, Hongkui He, Hailin Huang, Jiexin Wang, and Yi Cai. 2025. Rethinking-based Code Summarization with Chain of Comments. In *Proceedings of the 31st International Conference on Computational Linguistics*. 3043–3056.
[14] Sicong Cao, Xiaobing Sun, Ratnadira Widyasari, David Lo, Xiaoxue Wu, Lili Bo, Jiale Zhang, Bin Li, Wei Liu, Di Wu, et al. 2025. A Systematic Literature Review on Explainability for ML/DL-based Software Engineering. *Comput. Surveys* 58, 4 (2025), 1–34.
[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[16] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering* 48, 12 (2021), 4818–4837.
[17] Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the Effectiveness of LLM-as-a-judge for Code Generation and Summarization. *IEEE Transactions on Software Engineering* (2025).
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.
[19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
[20] Minying Fang, Xing Yuan, Yuying Li, Haojie Li, Chunrong Fang, and Junwei Du. 2025. Enhanced Prompting Framework for Code Summarization with Large Language Models. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1630–1653.
[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[22] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Shaomeng Cao, Kechi Zhang, and Zhi Jin. 2023. Interpretation-based code summarization. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. IEEE, 113–124.
[23] GitHub. 2025. *GitHub Copilot.* https://github.com/features/copilot
[24] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers.

[25] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[27] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[28] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[29] Miriam Horovicz and Roni Goldshmidt. 2024. TokenSHAP: Interpreting large language models with Monte Carlo shapley value estimation. In *Proceedings of the 1st Workshop on NLP for Science (NLP4Science)*. 1–8.

[30] Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, et al. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905* (2024).

[31] Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, et al. 2025. Opencoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 33167–33193.

[32] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[33] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *ArXiv* (2019).

[34] Dahyeon Jeong, Shilpa Aggarwal, Jonathan Robinson, Naresh Kumar, Alan Spearot, and David Sungho Park. 2023. Exhaustive or exhausting? Evidence on respondent fatigue in long surveys. *Journal of Development Economics* 161 (2023), 102992.

[35] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[36] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*. 1646–1656.

[37] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[38] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. 2024. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124* (2024).

[39] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[40] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).

[41] Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 47–51.

[42] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[43] Johan Linaker, Sardar Muhammad Sulaman, Martin Höst, and Rafael Maiani de Mello. 2015. Guidelines for conducting surveys in software engineering v. 1.1. *Lund University* 50 (2015), 1–64.

[44] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).

[45] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[46] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[47] Qing Lyu, Marianna Apidianaki, and Chris Callison-Burch. 2024. Towards faithful model explanation in nlp: A survey. *Computational Linguistics* 50, 2 (2024), 657–723.

[48] Vladimir Makharev and Vladimir Ivanov. 2025. Code Summarization Beyond Function Level. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 153–160.

[49] Ričards Marcinkevičs and Julia E Vogt. 2020. Interpretability and explainability: A machine learning zoo mini-tour. *arXiv preprint arXiv:2012.01805* (2020).
[50] Melkamu Mersha, Khang Lam, Joseph Wood, Ali K Alshami, and Jugal Kalita. 2024. Explainable artificial intelligence: A survey of needs, techniques, applications, and future direction. *Neurocomputing* 599 (2024), 128111.
[51] Nicholas Metropolis and Stanislaw Ulam. 1949. The monte carlo method. *Journal of the American statistical association* 44, 247 (1949), 335–341.
[52] Ran Mo, Dongyu Wang, Wenjing Zhan, Yingjie Jiang, Yepeng Wang, Yuqi Zhao, Zengyang Li, and Yutao Ma. 2025. Assessing and analyzing the correctness of github copilot's code suggestions. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–32.
[53] D OBrien, S Biswas, SM Imtiaz, R Abdalkareem, E Shihab, and H Rajan. [n. d.]. Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. In 2024 IEEE/ACM 46th International Conference on Software Engineering (Piscataway, NJ, April 14-April 20 2024). *IEEE, p. to appear* ([n. d.]).
[54] OpenAI. 2024. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/ https://openai.com/index/gpt-4-1/.
[55] David N Palacio, Dipin Khati, Daniel Rodriguez-Cardenas, Alejandro Velasco, and Denys Poshyvanyk. 2025. On Explaining (Large) Language Models For Code Using Global Code-Based Explanations. *arXiv preprint arXiv:2503.16771* (2025).
[56] David N Palacio, Daniel Rodriguez-Cardenas, Alejandro Velasco, Dipin Khati, Kevin Moran, and Denys Poshyvanyk. 2024. Towards more trustworthy and interpretable LLMs for code through syntax-grounded explanations. *arXiv preprint arXiv:2407.08983* (2024).
[57] David Nader Palacio, Alejandro Velasco, Nathan Cooper, Alvaro Rodriguez, Kevin Moran, and Denys Poshyvanyk. 2024. Toward a theory of causation for interpreting neural code models. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1215–1243.
[58] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[59] Lev Pevzner and Marti A Hearst. 2002. A critique and improvement of an evaluation metric for text segmentation. *Computational Linguistics* 28, 1 (2002), 19–36.
[60] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
[61] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.
[62] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[63] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.
[64] Lloyd S Shapley et al. 1953. A value for n-person games. (1953).
[65] Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. 2022. An exploratory study on code attention in BERT. In *Proceedings of the 30th ieee/acm international conference on program comprehension*. 437–448.
[66] Ensheng Shi, Yanlin Wang, Fengji Zhang, Bei Chen, Hongyu Zhang, Yanli Wang, Daya Guo, Lun Du, Shi Han, Dongmei Zhang, et al. 2025. SoTaNa: An Open-Source Software Engineering Instruction-Tuned Model. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 1–12.
[67] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–119.
[68] Ankita Nandkishor Sontakke, Manasi Patwardhan, Lovekesh Vig, Raveendra Kumar Medicherla, Ravindra Naik, and Gautam Shroff. 2022. Code summarization: Do transformers really understand code?. In *Deep Learning for Code Workshop*.
[69] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959* (2024).
[70] Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. 2025. The future of AI-driven software engineering. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–20.
[71] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[72] Yuvraj Virk, Premkumar Devanbu, and Toufique Ahmed. 2025. Calibration of Large Language Models on Code Summarization. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2944–2964.
[73] Antonio Vitale, Antonio Mastropaolo, Rocco Oliveto, Massimiliano Di Penta, and Simone Scalabrino. 2025. Optimizing Datasets for Code Summarization: Is Code-Comment Coherence Enough? *arXiv preprint arXiv:2502.07611* (2025).
[74] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[75] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards understanding the characteristics of code generation errors made by large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 717–717.
[76] Martin Weyssow, Aton Kamanda, Xin Zhou, and Houari Sahraoui. 2024. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences. *arXiv preprint arXiv:2403.09032* (2024).
[77] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 196–202.
[78] Jie Wu, Haoling Li, Xin Zhang, Xiao Liu, Yangyu Huang, Jianwen Luo, Yizhen Zhang, Zuchao Li, Ruihang Chu, Yujiu Yang, et al. 2025. Teaching your models to understand code via focal preference alignment. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 14014–14034.
[79] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.
[80] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675* (2019).
[81] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology* 15, 2 (2024), 1–38.
[82] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).