POLITECNICO DI TORINO
Repository ISTITUZIONALE

Beyond syntax: enhancing automated documentation with data differences

Original
Beyond syntax: enhancing automated documentation with data differences / Fantino, G., Vetro', A., Torchiano, M., Cappelluti, F.. - In: AUTOMATED SOFTWARE ENGINEERING. - ISSN 0928-8910. - ELETTRONICO. - 33:3(2026). [10.1007/s10515-026-00623-y]

Availability:
This version is available at: 11583/3010518 since: 2026-05-27T15:19:34Z

Publisher:
Springer

Published
DOI:10.1007/s10515-026-00623-y

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

26 November 2026

Automated Software Engineering (2026) 33:77
https://doi.org/10.1007/s10515-026-00623-y

Beyond syntax: enhancing automated documentation with data differences

Giacomo Fantino[1] · **Antonio Vetro'**[1] · **Marco Torchiano**[1] · **Federica Cappelluti**[2,3]

Received: 17 July 2025 / Accepted: 4 April 2026
© The Author(s) 2026

Abstract

Modern software development automation is mostly based on AI, covering every aspect of code production and maintenance, throughout the entire software development lifecycle, from requirements and code writing to testing and maintenance. Code commenting is no exception. Automated code comment generation methods rely on static syntactic and lexical features of source code. However, these approaches frequently underperform in data-centric software applications, where understanding the effect of code on data is essential. We explore an execution-aware extension to automatic documentation generation. In this exploratory work, we aim at capturing post-execution data transformations (i.e., *semantic data differences)* that reveal the code's effect on data, and use it as a complementary signal alongside existing code representations to automate explanatory comments for data wrangling code. We build a curated dataset of Python notebooks from Kaggle and apply a lightweight execution tracer to extract structured descriptions of runtime data transformations. We define a formal grammar for capturing these effects and integrate them into a multimodal encoder-decoder model using co-attention mechanisms. Multiple training strategies are explored to assess the impact of this new modality on comment generation. Our evaluation reveals that models incorporating this modality performed competitively with code-only baselines. Notably, in cases where no observable data transformation occurred, the presence of symbolic $\langle \text{no_diff} \rangle$ signals led to improved robustness and higher comment quality, as measured by both automatic and human evaluation metrics. However, we did not observe improvements in comment quality in semantically rich scenarios, suggesting possible paths of improvement for future research direction. Qualitative analysis of generated comments supports this pattern, indicating that the modality helps stabilize comments by reducing unnecessary or speculative details in neutral cases, but does not provide yet consistent guidance when meaningful data transformations occur. These trends are less pronounced on a larger, noisier extended test set, suggesting sensitivity to comment–code alignment. Our study demonstrates the feasibility and

Extended author information available on the last page of the article

Published online: 04 May 2026

potential of using execution-derived feedback as a complementary signal in automated comment generation. While the current approach is limited by dataset size and modality noise, it demonstrates that post-execution state changes can guide more context-aware and stable code summarization. This suggests a promising direction for execution-sensitive models in assisting data-centric software development and its documentation.

Keywords Automated comment generation · Data wrangling · Machine learning · Multimodal learning · Runtime code analysis

1 Introduction

With the advent of Generative AI, automation of software engineering processes and artifacts have become a widespread practice (Ozkaya 2023), impacting different activities throughout the software development lifecycle (Şimşek et al. 2024). About 4 out of 5 respondents to the the GitLab's Global DevSecOps Report reported that they are using AI in software development or plan to in the next 2 years (it was 3 out of 5 in 2023)[1]. Automation and AI assistance are relevant to almost every aspect of software development, ranging from writing requirements (Kretzer et al. 2025) to writing code (Bird et al. 2023) and reviewing it (Davila et al. 2024), code translation (Dhruv and Dubey 2025), testing (Treshcheva et al. 2025), and many other activities.

Although the rapid emergence of GenAI in software engineering has raised several issues, including security (Kudriavtseva et al. 2025), licensing and copyright (Stalnaker et al. 2025), psychological burdens on developers (Meem and Johnson 2025), threats to creativity (Jackson et al. 2025, and a variety of ethical issues such as transparency, fairness and privacy (Donvir and Sharma 2025), expectations and market forecasts for Gen-AI based automation in software engineering are generally very positive[2].

This is happening in a world where data has become an inseparable element of software development, and data-centric applications are widespread. A plethora of new platforms are dedicated to hosting scripts, analyses and, more generally, code for processing data. In recent years, interactive computational notebooks have become the de facto medium for data analysis, machine learning prototyping, and scientific experimentation (Mondal et al. 2023; Schröder et al. 2019). Platforms like Jupyter are especially prevalent in data-centric workflows where code is deeply intertwined with dataframes, visualizations, and exploratory narratives. Despite their ubiquity, such notebooks often suffer from sparse or low-quality inline documentation, particularly in the context of data wrangling code (Mondal et al. 2023), i.e. code that transforms, selects, cleans, or reshapes raw data into forms suitable for analysis, modeling, or

[1] See https://about.gitlab.com/developer-survey/, last visited on 14 July 2025

[2] See for example https://www.researchandmarkets.com/reports/6014321/generative-artificial-intelligence-ai-in (last visited on 14 July 2025), whose prediction is that the market will be three times the size it is now by 2029

visualization (Huang et al. 2024). This lack of clarity poses a significant barrier to understanding, reuse, and reproducibility (Hu et al. 2022).

This aspect is crucial for modern data-centric software applications – which are becoming increasingly prevalent, as discussed above – and in particular for research software, where code readability and clarity are essential for validating and building upon experimental results (Wilkinson et al. 2016). As research communities increasingly embrace open science principles, the quality of code documentation becomes more than a usability issue: it is a prerequisite for transparent and sustainable scientific workflows. In general, well-documented code is important for software developers because it facilitates smooth and correct adaptation in the receiving environment, thereby increasing understandability and potential for reuse.

Existing techniques for automated comment generation are largely based on the static syntactic structure of source code, such as abstract syntax trees (ASTs) or token sequences. These approaches often overlook the semantic effect that code has on the data it manipulates, a crucial aspect in data-centric programming. This disconnect is especially pronounced in data wrangling code where simple-looking operations can produce non-trivial transformations in data structures, shapes, and semantics. In fact, in many software applications, to understand code it is first necessary to understand data and how code transforms it.

In this paper, we explore a novel direction in automatic code documentation: we introduce semantic data differences, a novel auxiliary modality that captures the semantic changes in structured data, the transformations observed after the execution of a code cell in the data. These semantic changes are later formalized as structured descriptors representing high-level data transformations. Rather than relying solely on surface-level code features, we analyze post-execution data state changes, such as variable additions or structural modifications to dataframes, including added or removed columns, and represent these changes as structured natural language descriptors (e.g., `<modified> df <added_col> log_x <added> model`). This additional modality provides concrete, context-sensitive signals about the effect of code, which can be leveraged to generate more accurate, interpretable, and task-relevant comments. It extends existing representations by adding lightweight, execution-derived cues about how code affects data. Rather than replacing syntactic information, it provides complementary context that may support models in capturing code intent, particularly for data-centric workflows.

Our overarching goal in this work is to evaluate the utility of this new modality for enhancing comment generation models in data-centric coding environments. To this end, we make the following contributions:

- we define a structured representation of data differences as a source of semantic context for code interpretation;
- we curate and openly share a novel dataset of executed Python notebooks from Kaggle, enriched with tracked data transformations and corresponding comments[3];

[3] Link to https://zenodo.org/records/15985078.

- we implement a prototype encoder that integrates both code and data difference signals, exploring alternative pretraining strategies;
- we conduct an empirical evaluation to assess the quality and informativeness of the generated comments, both via automatic metrics and human judgments;
- we discuss implications for automated documentation and the broader landscape of open, reproducible research on software engineering.

By explicitly modeling the interaction between code and the data it modifies, this work provides initial empirical evidence for context-aware code understanding and offers a concrete step toward more intelligent tooling for scientific software development.

The remainder of the paper is organized as follows. In Section 2, we summarize the state of the art in automated comment generation and motivate the research idea. Then, in Section 3 we introduce our formal definition of semantic data difference with a structured grammar. In Section 4 we detail a proposed multimodal model architecture implementing our novel modality, while also outlining the construction of our dataset, including the pipeline used to extract execution-derived transformations from computational notebooks. Section 5 presents the experimental setup and research questions, followed by Section 6, where we report and analyze both automatic and human evaluation results. In Section 7, we reflect on the implications of our findings and identify promising directions for future research. We discuss threats to validity in Section 8 and conclude the paper in Section 9. Three appendices are available: Appendix A reports additional automatic evaluation results on an extended test set, providing further evidence of the generalizability of our findings, Appendix B presents a 5-fold cross-validation analysis that demonstrates the stability and robustness of the training procedure, and Appendix C reports an ablation study on runtime value-augmented signals to assess the impact of enriching the semantic modality with additional execution detail.

2 Motivation and background

2.1 Related work

To improve contextual awareness, some methods extend the input space to include neighboring cells in computational notebooks, variable usage patterns across code blocks or the project context of the source code (Haque et al. 2020; Zhou et al. 2022; Bansal et al. 2021). These approaches emphasize variable and cell dependency but often remain grounded in static-based analyses. While such techniques provide improved local context, they do not typically capture the semantic effect of code on data content, particularly when transformations occur dynamically at runtime.

A parallel line of research has explored dynamic analysis for aiding human understanding in data-centric workflows. WrangleDoc (Yang et al. 2021) instruments the execution of Jupyter notebooks to produce descriptive, human-readable summaries of data transformations, including representative examples and domain-specific annotations. It does not generate comments or explanatory summaries; rather, it visualizes changes in variable state during execution, similar to variable-inspector or debug-

ging features found in IDEs. Similarly, Patterson et al. (2019) proposed a system that builds semantically enriched flow graphs based on runtime traces and domain-specific ontologies. Their work focuses on transforming low-level traces into semantically meaningful nodes such as “fit-model” or “load-data,” primarily for human understanding. These summaries are designed for human consumption, leveraging illustrative examples and narrative explanations. While such human-oriented systems demonstrate the value of leveraging execution traces to clarify the effects of data-wrangling code, they do not provide machine-readable representations nor are they intended for integration into automated generation pipelines. In a more formal direction, Data Diff (Sutton et al. 2018) introduces interpretable summaries ("patches") to capture distributional and formatting changes between repeated data samples, but remains focused on human-readable explanations rather than machine-consumable representations. In contrast, prior work has demonstrated that symbolic representations of runtime behavior can be systematically captured and leveraged for the detection of software bugs and branch coverage (Dimitrov and Zhou 2007; Ding et al. 2024). Recent approaches have begun to incorporate runtime execution information into LLM-based code understanding, teaching LLMs to reason about program execution using inline execution traces and execution-aware chain-of-thought rationales (Ni et al. 2024).

2.2 Limitations of static code understanding

While all the aforementioned methods are effective for capturing the syntactic or structural properties of code, they often fall short in scenarios where understanding the semantic impact of code execution is essential. This is particularly acute in data-centric programming environments. In these contexts, code is rarely an end in itself; rather, its primary purpose lies in transforming data through a series of wrangling, filtering, and feature engineering operations. Such transformations frequently exhibit semantics that are not readily inferable from syntax alone. For example, a single line of code may implicitly add a derived column, change the distribution of a dataset, or instantiate a model object. These operations have significant implications for downstream analysis, but with minimal lexical trace. Moreover, prior work has shown that documentation quality in notebook-based workflows tends to be sparse or low quality, especially for intermediate transformation steps (Mondal et al. 2023). This represents a critical barrier to reproducibility, collaboration, and code reuse.

2.3 A new direction: introducing semantic data differences

Drawing inspiration from the notions of runtime-informed symbolic abstraction, our approach extends its applicability to source code modeling in multimodal learning frameworks. Specifically, we repurpose semantic transformations derived from code execution into structured input representations, thereby enriching model understanding of code functionality beyond its surface-level syntax. We introduce semantic data differences as an auxiliary input modality for comment generation. These differences are defined as structured, symbolic representations of high-level transformations observed during code execution. Their contribution is expected to be complemen-

tary: static code features supply the core syntactic representation, whereas execution-derived cues introduce additional semantic signals that can enrich the model's understanding of data-centric operations. Our aim is to generate more accurate, context-aware and semantically grounded comments by modelling post-execution data semantics alongside the code. The main source of inspiration of the proposed approach is the work of Yang et al. (2021), previously described in Section 2.1. However, the intent and implementation of our approach differ substantially: rather than summarizing code effects for human readers, we treat execution-derived semantic transformations as machine-readable inputs for a multimodal encoder model. These structured data difference descriptors are learned jointly with code representations to inform the generation of explanatory comments. While our formal grammar for expressing data differences draws conceptual inspiration from prior efforts (Yang et al. 2021; Patterson et al. 2019), its use as a modeling signal, rather than a presentation aid, marks a key distinction. Thus, the resulting representations are optimized for compactness, compositionality, and alignment with code tokens, rather than narrative expressiveness. This distinction positions semantic data differences not as a documentation system in themselves, but as a novel modeling signal for automated comment generation.

This work represents the first attempt to operationalize post-execution data semantics as an auxiliary modality in neural comment generation, an underexplored yet critical dimension. In doing so, it introduces a novel axis of representation, complementary to syntax and data dependencies, particularly well-suited to the interpretive challenges of data-centric software.

3 Proposed approach

Understanding code in data-centric software requires more than parsing syntax: it demands awareness of how code execution transforms data. In particular, the functional intent of data wrangling code, such as filtering rows, adding columns, or reshaping dataframes, is often implicit and not readily captured through static analysis. To identify such code, we adopt an execution-based criterion: a code cell is identified as data-wrangling if it produces observable semantic modifications to data structures. Conversely, cells that do not induce detectable changes, regardless of their syntactic content, can be treated as non-data-wrangling. While this approach may occasionally under-identify cases where syntactic operations do not yield effective transformations (e.g., due to conditional logic or redundant actions), it reliably captures the majority of meaningful modifications in practical workflows.

To address the implicit nature of such transformations, we introduce a structured method to explicitly describe these effects, **semantic data differences**, defined as a structured, semantic description of how variables and data structures change as a result of executing a code cell. These descriptions abstract away from raw values and instead focus on high-level transformations.

To effectively integrate post-execution data semantics into learning-based models, it is necessary to represent such transformations in a structured and machine-readable form. Rather than relying on verbose or noisy natural language or raw value compari-

sons, we adopt a symbolic representation that enables alignment with code tokens, facilitates model training, and captures the semantic intent of runtime effects. Thus, we define a lightweight, structured grammar that abstracts the high-level effects of code execution on program state. This grammar encodes what changes occur to variables and data structures, rather than how these changes are implemented at the syntactic level. We define a data difference as a sequence of atomic semantic operations expressed in the following formal grammar:

$$
\begin{aligned}
\mathcal{G} \quad ::= \quad & \langle \mathsf{no_diff} \rangle \\
| \quad & \langle \text{added} \rangle \; v \\
| \quad & \langle \text{removed} \rangle \; v \\
| \quad & \langle \text{modified} \rangle \; v \\
| \quad & \langle \text{modified} \rangle \; df \; \langle \mathsf{added_col} \rangle \; c \\
| \quad & \langle \text{modified} \rangle \; df \; \langle \mathsf{removed_col} \rangle \; c \\
| \quad & \mathcal{G} \; \mathcal{G}
\end{aligned}
$$

Where *v* denotes a generic variable, *df* denotes a DataFrame variable and *c* denotes a column identifier within a DataFrame.

Each code cell may produce a sequence of such difference tokens, reflecting multiple changes during execution. For instance:

<modified> df <added_col> log_x <added> model

indicates that a DataFrame variable df was modified (e.g., transformed), a column 'log_x' was added to it, and a new variable model was initialized. In the absence of detectable changes, we emit a special token ⟨no_diff⟩ to signify that the cell did not alter program state in a semantically relevant way (e.g., plotting, printing, or imports).

This symbolic format offers the following advantages:

- Compactness: It encodes semantic effects succinctly, avoiding verbose or redundant natural language and reducing sequence length for transformers.
- Interpretability: Tokens are interpretable by both humans and models (aiding during debugging and training supervision) and can be composed to represent complex transformations.
- Generality: Although tailored to common data-wrangling operations (e.g., on DataFrames), the grammar is extensible to other data types or operations, enabling transferability across datasets or frameworks.

This representation is not intended to capture the precise before-and-after values of variables, but rather to encode the nature of change in a form that is both interpretable and generalizable. The data difference acts as a bridge between code execution and natural language summarization, by making explicit the semantic operations implied by the code: filtering, feature construction, model instantiation, and so forth. In preliminary experiments, we also explored augmenting the data difference sequence with runtime outputs using lightly processed variable-update messages (e.g., <modified> var <val> old_val → new_val). However, these signals introduced

substantial noise relative to the scale and structure of our dataset. The resulting variability in message phrasing and verbosity negatively affected the encoder's ability to form stable representations, ultimately leading to a degradation in comment quality across both automatic and human evaluation. These observations reinforced our choice of using a compact, grammar-based abstraction, which provides consistent and machine-readable cues while avoiding the brittleness associated with raw or semi-structured natural-language runtime messages.

This modality shifts the focus of comment generation from merely analyzing the syntactic structure of code to understanding its semantic effect on data. By combining lightweight data profiling with runtime introspection, we extract signals that align with human notions of intent, signals typically inaccessible to purely lexical or syntactic models. Semantic data differences thus provide a principled mechanism for grounding comment generation in dataflow and program state, enabling richer and more context-aware summarization.

The formal grammar we propose draws inspiration from prior work such as WrangleDoc (Yang et al. 2021) and Data Diff (Sutton et al. 2018), which aim to produce human-readable summaries of data transformations. In contrast, our design prioritizes integration with neural architectures: the grammar functions as a structured token sequence optimized for deep learning, favoring compactness, compositionality, and alignment over natural language fluency. This representation constitutes the core of the additional semantic modality introduced in our multimodal comment generation framework, detailed in the following section. This design choice also fixes the level of abstraction of the semantic signal. Operations such as row filtering, type casting, or aggregation are represented by a coarse descriptor such as $\langle \text{modified} \rangle$ or $\langle \text{added} \rangle$ when they are followed by assignment rather than a transformation-specific token; when they are applied by updating a DataFrame, the same modification are represented by column-level markers (e.g., $\langle \text{modifiedcol} \rangle$ or $\langle \text{addedcol} \rangle$). Notably, row-level operations (e.g., filtering) are captured indirectly by our strategy: because the comparison is performed at the column-content level, these operations change the contents of every column. As a result, all columns are detected as modified, yielding a sequence of $\langle \text{modifiedcol} \rangle$ tokens, one for each column. Therefore, distinct wrangling actions can map to the same symbolic sequence, which helps keep the representation compact and consistent, while motivating future extensions toward higher-resolution transformation types.

4 Method

4.1 Cross-modal alignment via co-attention

To integrate syntactic and semantic signals from the source code and its associated data difference descriptor, we adopt a co-attention mechanism that allows fine-grained interactions between the two input modalities. Unlike self-attention (Vaswani et al. 2017), which models internal dependencies within a single sequence, co-attention enables each token in one modality to attend to all tokens in the other, allowing the model to capture cross-modal relevance patterns (Lu et al. 2019).

Formally, defining $C \in R^{n \times d}$ the contextual embeddings of the source code and $D \in R^{m \times d}$ the contextual embeddings of the data difference, where n and m are the number of tokens and d is the hidden dimensionality, the co-attention module computes attention weights between all code–data token pairs using scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{1}$$

Conditioning one modality with the tokens of the other. In our setup:

- code embeddings are conditioned on data-diff embeddings: $\text{Attention}(C, D, D)$
- data-diff embeddings are conditioned on code embeddings: $\text{Attention}(D, C, C)$

The use of co-attention in our architecture is motivated by the need to model fine-grained interactions between syntactic code tokens and the semantic signals encoded in data difference descriptors. Unlike simple concatenation or late fusion techniques, co-attention layers enable each token in one modality to dynamically attend relevant elements in the other, fostering context-aware alignment across modalities. This is particularly important in our setting, where data difference sequences are typically compact but semantically dense, while code tokens are structurally rich but ambiguous in isolation. By facilitating bi-directional interaction, co-attention helps the model disambiguate code intent based on observed effects.

Our adoption of co-attention also draws conceptual support from its effectiveness in other domains, such as vision-language tasks (Lu et al. 2019), session-based recommendation (Chen and Chen 2021), and aspect-level sentiment analysis (Liu et al. 2021). This demonstrates the general utility of co-attention in aligning heterogeneous but complementary inputs, supporting its relevance for our multimodal setup involving source code and semantic execution feedback.

4.2 Model architecture

We propose the multimodal architecture shown in Fig. 1, that integrates both the syntactic structure of code and its semantic effect on data at runtime. To achieve this, the model encodes each modality independently before fusing them through cross-modal interactions, allowing comment generation to be conditioned on both the code's structural form and the transformations it induces. Specifically, we introduce a novel

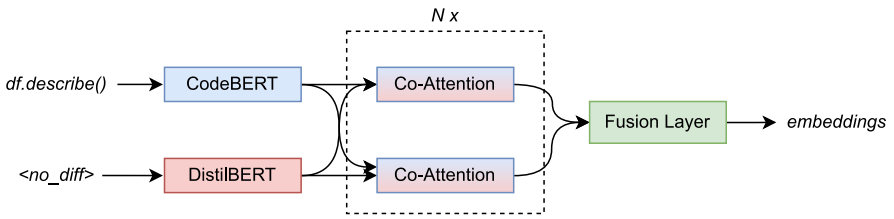

Fig. 1 Model architecture

encoder for data differences that processes structured semantic descriptors derived from code execution (as defined in Section 3). Each encoder transforms its respective input, source code and data differences, into contextual embeddings. These are then passed through three cross-attentional layers that align and co-represent features across modalities. A final fusion module merges the representations into a unified embedding used for downstream comment generation.

Our architecture builds on an encoder-decoder framework, with the encoder initialized from CodeBERT (Feng et al. 2020), a transformer-based model pre-trained on source code and natural language. Specifically, we leverage CodeBERT's initialization on the CodeXGLUE dataset (Lu et al. 2021), where it was trained for Python code summarization. For the data difference modality, instead, we use the standard DistilBERT weights as initialization (Sanh et al. 2019). Although transformer encoders pretrained on programming languages–such as CodeBERT–have shown strong performance on NL–PL tasks, they are not suitable for modeling semantic data difference sequences. Their pretraining objectives and token distributions are tailored to source-code syntax and identifier-level regularities, whereas our data-diff representation is a symbolic state-transition sequence that diverges substantially from typical code corpora. Using a code-pretrained encoder for this modality would therefore introduce a significant domain shift, limiting the meaningful reuse of pretrained weights and reducing efficiency given the size of our curated dataset. Initializing the data encoder from a general-purpose language model and pretraining it directly on data-diff sequences ensures alignment with the structure of this modality and provides a more effective starting point for downstream learning. A definitive assessment of alternative initializations would require dedicated empirical study, including training strategies specifically tailored to the data-diff modality, which we leave for future work.

Training proceeds in two stages: pre-training and finetuning (Fig. 2). During pretraining, step (b) in Fig. 2, we initialize the dual-encoder model and train it using a masked language modeling (MLM) objective applied across both code and data difference modalities. Specifically, the model learns to reconstruct masked tokens in either the code snippet or the associated data difference sequence, while condi-

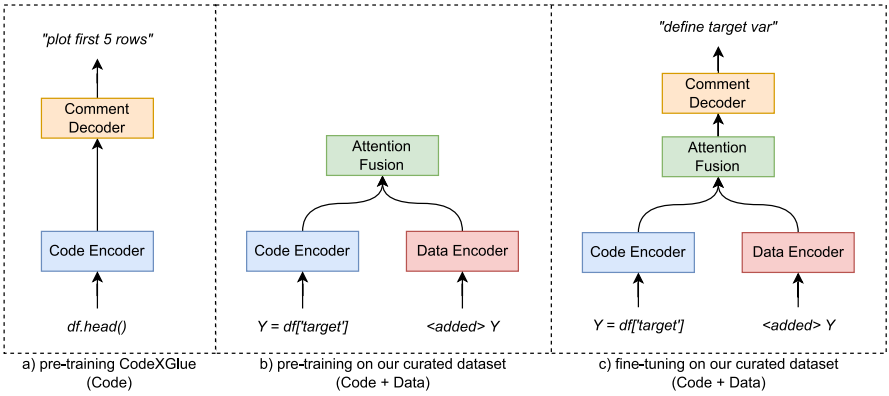

Fig. 2 Training procedure for the multimodal model, with examples of comments in input and output

tioning on both inputs. This cross-modal pretraining objective encourages the model to learn the correspondences between code tokens and the semantic signal encoded in the associated data-difference sequence. Importantly, the pretraining is conducted solely on our custom dataset, as the presence of the data difference signal is essential to this objective. To investigate the influence of different training strategies and data configurations, including how such design choices impact the model's ability to learn from cross-modal differences, we evaluate the following three variants of our model, each designed to explore specific aspects of cross-modal representation learning. A visual comparison of these variants is presented in Table 1, highlighting differences in encoder initialization, inclusion of ⟨no_diff⟩ samples during pre-training (to ensure a balanced learning signal), and masking strategies applied during pretraining.

Data-Only Pretrain (*"NoDiff-Omitted"*): In this configuration only the data difference encoder was pretrained, while the code encoder was kept frozen during this stage. Furthermore, training data containing the special ⟨no_diff⟩ token was excluded from pretraining. This variant allows us to assess whether isolated semantic pretraining contributes useful inductive biases, particularly in capturing structural patterns in data transformation.

Code-Data Joint Pretrain Joint Pretrain (*"20% NoDiff"*): We simultaneously pretrain both the data and code encoders using a joint masked language modeling objective. Importantly, to mitigate distributional skew and overfitting to uninformative samples, we constrain the proportion of ⟨no_diff⟩ samples to 20%, randomly reshuffled at each epoch.

Code-Data Joint Pretrain Joint Pretrain with Emphasis on Data Tokens (*"Masked-Data Boost"*): The final variant builds upon *20% NoDiff* by modifying the masking strategy. Specifically, we double the masking probability for tokens in the data difference sequence, raising it from 15% to 30%, to encourage the model to focus more on data semantics during reconstruction.

In step (c) of Fig. 2, we extend the pretrained encoder into a full encoder-decoder architecture by attaching the decoder of the CodeBERT-based model pretrained on CodeXGLUE. The combined model is fine-tuned on our dataset of source code, data differences and human-written comments. This phase aims to optimize generation quality while preserving the cross-modal representations learned during pretraining.

All experiments were conducted on a workstation equipped with an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 16GiB of RAM, and an NVIDIA GeForce RTX 2070 GPU (8GiB VRAM). Due to GPU memory limitations, we employed gradient accumulation during pre-training, effectively simulating larger batch sizes by accumulating gradients over multiple forward passes. The system ran Pop!_OS 22.04 LTS as the operating system. We used Python 3.10.12 and PyTorch 2.6.0 with CUDA 12.7 support for model development and training. Pre-training was conducted for up to 10 epochs using a batch size of 4, while fine-tuning was performed for 5

Table 1 Comparison of model variants in pretraining configuration

Model Variant	Code Encoder	Data Encoder	<no diff>	Masking Rate
NoDiff-Omitted	Frozen	Trained	Excluded	15%
20% NoDiff	Trained	Trained	20%	15%
Masked-Data Boost	Trained	Trained	20%	**30%on data**

epochs with a batch size of 8. For both stages, the AdamW optimizer was used with a learning rate of 5e-5, weight decay of 0.0, and epsilon of 1e-8. During pre-training, we monitored both training and validation loss; if validation loss plateaued while training loss continued to decrease, training was stopped early to mitigate overfitting. At inference time, we employed beam search decoding with a beam width of 5 to enhance the quality of generated comments, following the approach used in the original CodeBERT architecture (Feng et al. 2020). A full list of Python dependencies is provided in the accompanying reproducibility package.

4.3 Training set construction

To support the evaluation of our proposed multimodal model, we introduce a new dataset specifically designed to capture the interplay between code, data transformations, and explanatory comments. We use real-world notebook environments because they are specifically designed for data-wrangling applications and are widely available on code hosting platforms. The dataset consists of executed Python notebooks, enriched with tracked variable-level and data structure changes, extracted from live runs. The pipeline consists of 3 main steps as shown in Fig. 3: notebook collection, execution and dataset creation.

We collected publicly available notebooks from the Kaggle platform, with a primary focus on those submitted to data science competitions. These notebooks typically contain rich examples of data preprocessing, feature engineering, and model development, particularly for tabular datasets and exploratory workflows. Following prior work (Yang et al. 2021), we selected a subset of public datasets and retrieved the highest-voted notebooks associated with them. We specifically focused on notebooks for three key reasons. First, notebooks are inherently data-centric and reflect a step-by-step, exploratory coding style, where each code cell often performs meaningful data transformations. This makes them ideal for isolating and analyzing the semantic effect of individual operations on data structures. Second, unlike traditional scripts or repositories, Kaggle notebooks are typically self-contained and executable, as their required datasets are specified via metadata. This allowed us to reliably execute the notebooks, observe runtime effects, and extract structured data difference descriptors from each cell. Third, the cell-based execution model aligns with our goal of generating fine-grained, execution-aware code summaries, as it enables semantic tracking at the level of individual transformations. These characteristics make notebooks the most suitable source for building a dataset grounded in post-execution data semantics.

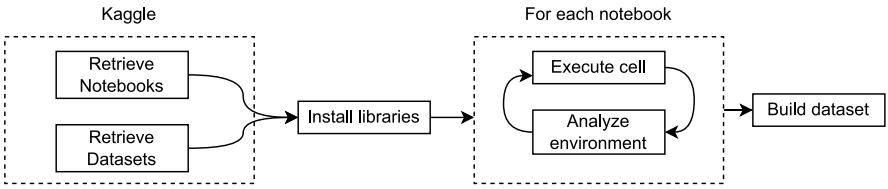

Fig. 3 Pipeline used to build the dataset

To maximize executability, we examined the metadata files accompanying each notebook, which specify the required datasets. When a referenced dataset was not included in the initial download, we retrieved it separately to ensure completeness and reproducibility. We also collected and installed the required libraries of all notebooks in a virtual environment. When conflicts between libraries were raised, typically due to incompatible version requirements for shared dependencies, we dropped the libraries used by fewer notebooks to maximize executability.

Each notebook was then executed in a controlled environment. During execution, we applied a custom instrumentation layer to log variable assignments, modifications, and data transformations, with a special focus on operations affecting dataframes. From these traces, we generated corresponding data difference descriptors (as defined in Section 3), which summarize the semantic effect of each cell's execution. Importantly, many contain in-cell comments. In these situations we parse the cell and subdivided it as multiple cells considering the comments. This maximized the number of samples we could retrieve from a single notebook. Following the methodology of Mondal et al. (2023), we also used the Spacy model (Spacy 2017) to summarize comments when those exceeded a threshold length, to mitigate the issue of long, noisy comments. Table 2 shows the statistics of the final extracted dataset, obtained after executing 4869 notebooks. We note that the average number of data-diff tokens is higher in validation due to a small number of outlier samples with unusually long data-diff sequences, which skew the mean. However, percentile-based statistics (e.g., median, 25th and 95th percentile) indicate that the typical data-diff length is comparable across training and validation. For the remaining modalities (code and comment tokens), the distributions match closely across splits.

To encourage reuse and further expansion of the dataset, we provide the datasets, the set of retrieved notebooks, a pre-configured virtual environment specification and the execution and tracking pipeline used to extract code and data changes.

Table 2 Statistics of the training and validation datasets

Modality	Statistic	Train	Validation
Samples	# samples	25,573	6,394
	% ⟨no diff⟩	47.78%	47.75%
# Code tokens	Average	35.2	35.0
	25th percentile	9.0	9.0
	Median # tokens	20.0	20.0
	75th percentile	39.0	38.0
	95th percentile	105.0	109.3
# Data-Diff tokens	Average	13.6	80.5
	25th percentile	1.0	1.0
	Median # tokens	2.0	2.0
	75th percentile	4.0	4.0
	95th percentile	10.0	10.0
# Comment tokens	Average # tokens	12.5	12.8
	25th percentile	4.0	4.0
	Median # tokens	8.0	8.0
	75th percentile	17.0	17.0
	95th percentile	34.0	33.0

5 Experiment design

5.1 Research questions

To rigorously evaluate the effectiveness of the proposed data difference modality in enhancing automated comment generation, we designed a set of empirical experiments guided by the following research questions:

- RQ1: To what extent does the inclusion of data difference information improve the quality of generated comments when meaningful data transformations are present?
- RQ2: How does the presence of data difference information affect model performance in scenarios where no meaningful semantic transformation is detected?

Together, these questions enable a comparative analysis of model variants across two complementary subpopulations of the test set, reflecting different levels of semantic availability. While RQ1 aims to quantify the benefits of incorporating semantic feedback in cases where data transformations are detected, RQ2 is driven by a complementary concern: assessing the behavior of the model when no such transformations are observed. In these cases, the data difference sequence is reduced to a $\langle \texttt{no_diff} \rangle$ token, indicating that execution did not produce a measurable change in the tracked data structures. Thus, it is critical to evaluate whether the model remains robust and generalizable in the absence of explicit data changes.

Herein we report on the experimental setup, the test-set constructed and the selected metrics. The overall experimental process is depicted in Fig. 4.

5.2 Experimental setup

To assess the effectiveness of incorporating data difference information into comment generation, we designed a set of controlled experiments comparing our proposed multimodal model to strong code-only baselines. The experimental setup evaluates model performance across different data conditions and comment types, using both automatic metrics and human evaluations.

As a comparative baseline, we utilize a standard encoder–decoder architecture based solely on CodeBERT, called *Code-Only*, fine-tuned on the same dataset without access to data difference descriptors. This model represents the state-of-the-art for code-only summarization and serves as a control condition for isolating the impact

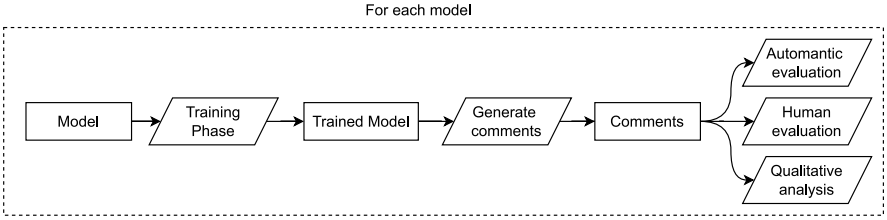

Fig. 4 Experimental process for answering the RQs

of the semantic modality. We also considered the pre-trained CodeBERT without fine-tuning on our dataset to verify the quality of the collected data, identified simply as *Pre-Trained*. To provide a stronger structural baseline that incorporates context beyond token sequences, we additionally include a GraphCodeBERT-based model in our comparison (Guo et al. 2021). GraphCodeBERT leverages data-flow edges and AST-level structure during pretraining, offering a richer structural prior than CodeBERT and enabling an assessment of the role of syntactic and data-flow information in code summarization. To ensure comparability with our Code-Only baseline, we adopted the same two-phase training strategy, but due to the model's sensitivity to encoder weights, we applied a conservative learning rate to the encoder during the CodeXGLUE phase to mitigate catastrophic forgetting, while allowing full optimization of the decoder.

Our proposed model extends *Code-Only* by incorporating the structured data difference descriptors via the multimodal architecture introduced in Section 4.2. The three multimodal variants are compared against the unimodal baselines and evaluated across both semantically rich and semantically neutral test subsets.

5.3 Test set

To construct a test set, we collected computational notebooks written by colleagues as part of their own research or embedded in the development of data-centric software. These notebooks were selected based on the quality of their in-line comments, ensuring that they both provide meaningful context for evaluating quality of comments, and capture the diversity of data wrangling tasks present in the broader dataset. This dataset supports both automated analysis and a complementary human evaluation process to assess the alignment between code and the associated generated comments.

To conduct the comparative evaluation, we partition the test set into two subsets based on the presence or absence of semantic execution feedback. The Data-Diff Present subset includes samples for which data difference information captures observable semantic transformations (RQ1). In contrast, the Data-Diff Absent subset includes samples with no detected semantic effects (RQ2), typically corresponding to cells performing logging, plotting, or import operations. Each model variant is evaluated independently on both subsets, enabling a population-level comparison of model behavior under varying semantic conditions and providing a principled foundation for evaluating the conditional utility of the proposed data difference modality across realistic usage scenarios. Table 3 shows the summary statistics of the two subsets. Noticeably, the average number of data diff tokens for the Data-Diff Absent subset is equal to 1 due to all samples presenting the single token $\langle \mathsf{no_diff} \rangle$.

Table 3 Statistics of the two test subsets

	Data-Diff Present	Data-Diff Absent
# samples	31	22
avg # code tokens per sample	28.1	17.3
avg # data-diff tokens per sample	4.6	1
avg # comment tokens per sample	5.6	6.5

5.4 Metrics

We evaluate the generated comments using a combination of automatically computed metrics and human assessment. For automatic evaluation, we employed the n-gram-based metrics commonly used to assess textual similarity: BLEU, METEOR, ROUGE-1, ROUGE-2 and ROUGE-L. While n-gram-based metrics provide insight into surface-level overlap, we complement them with embedding-based metrics such as CodeBERTScore and RoBERTa similarity to assess deeper semantic alignment, crucial for short, abstract code comments where exact lexical overlap may be sparse.

For human evaluation, we adopt a lightweight qualitative assessment protocol tailored to our specific domain of data-wrangling code in computational notebooks. While automatic metrics such as BLEU, METEOR, and CodeBERTScore offer a convenient way to evaluate textual similarity, they are known to poorly correlate with human preferences in the context of code summarization. Prior studies have shown that small improvements in these scores often fail to reflect genuine gains in summary quality (Roy et al. 2021). To capture such nuances, we include a complementary human evaluation phase to assess the informativeness, clarity, and alignment of generated comments with the underlying code intent. As our primary quantitative signal, we use a single holistic 3-point Likert scale that implicitly aggregates aspects such as relevance, fluency, and helpfulness. This choice keeps the annotation task lightweight and is well suited to our experimental context, where individual cells typically perform narrow, well-defined transformations. Annotators rate each comment on a scale from 1 (poor) to 3 (good), based on its correctness, clarity, and usefulness in describing the code's intent and its effect on data. A score of 3 is assigned when the comment is grammatically well-formed, semantically accurate, and provides meaningful insight into the code's effect. A score of 2 reflects comments that are partially correct or somewhat vague, but still helpful. A score of 1 is used for misleading, syntactically broken, or overly generic outputs.

To ensure contextual alignment, annotators were shown the original code and its generated comment side-by-side, and were instructed to focus on the usefulness of the comment in conveying the intent of the code block in a data-centric workflow. The three annotators, one PhD candidate and two university professors, all with multiple years of experience in Python and computational notebooks, independently evaluated the outputs using the shared rubric. In cases of disagreement, particularly when ratings varied substantially (e.g., scores of 1, 1, and 3), we held a joint review session to discuss and resolve the conflicts through consensus. This design reduces cognitive load and improves rating consistency, especially in small-scale evaluations. To assess the reliability of human evaluations, we computed Krippendorff's alpha on the ratings before and after the joint session (Hayes and Krippendorff 2007). The former reflects the initial independent judgments, while the latter quantifies the improvement in alignment obtained through discussion. Krippendorff's alpha is well-suited for ordinal data and accounts for chance agreement among raters. An alpha above 0.8 is typically considered strong agreement, while values between 0.667 and 0.8 are acceptable.

To further assess the qualitative characteristics of generated comments and to better understand the types of improvements or degradations induced by the semantic

modality, we conducted an additional fine-grained manual analysis on the generated comments. For each sample, we examined the generated comment and annotated it along four analytic dimensions commonly adopted in qualitative assessments of documentation quality (Dvivedi et al. 2024):

- Semantic correctness: whether the explanation reflects the true effect of the code.
- Syntactic correctness: grammatical well-formedness.
- Completeness: coverage of all relevant aspects of the code's behavior.
- Relevance: absence of superfluous or misleading information.

These codes allow us to capture different qualitative aspects of the generated comments that are not observable through automatic metrics or holistic scores. After coding, we aggregated results at model level for both the Data-Diff Present and Data-Diff Absent subsets, enabling a comparative interpretation of how the data difference modality affects different quality aspects of comments.

6 Results

In this section, we report the outcomes of our empirical evaluation, aimed at assessing the impact of incorporating semantic data differences into comment generation models for data-wrangling code. We present both automatic evaluation metrics and results from human annotation, following the experimental design described in Section 5. Additionally, Table 4 reports inter-rater agreement scores across models and test subsets before and after the joint discussion, measured with Krippendorff's α. The initial α values reflect the variability of independent ratings, whereas the post-discussion values show the increased alignment obtained through the resolution process. A few models exhibit initial α values below commonly accepted thresholds, which is consistent with the inherent subjectivity of the task during independent assessment. These discrepancies were substantially reduced following the consensus discussion.

For completeness, we also evaluated all models on a larger, automatically extracted test set. For reasons of space and focus, we report here only the results on the small curated test set; the corresponding results for the larger test set are provided in Appendix A, and their interpretation is addressed in the discussion in Section 7.

Table 4 Inter-rater agreement before and after the review session for each model and subset indicated with Krippendorff's alpha

	Before Discussion		After Discussion	
Model	Diff Present	Diff Absent	Diff Present	Diff Absent
Pre-Trained	0.41	0.63	0.60	0.69
Code-Only	0.60	0.75	0.66	0.84
AST-Graph	0.73	0.86	0.78	0.86
NoDiff-Omitted	0.75	0.77	0.75	0.77
20% NoDiff	0.77	0.63	0.81	0.70
Masked-Data Boost	0.86	0.76	0.86	0.76

6.1 RQ1: impact of semantic modality when data transformations are present

As reported in Table 5, in the presence of detectable data transformations, the baselines *Code-Only* and *AST-Graph*, fine-tuned only on source code, achieved the highest scores across most metrics, including BLEU, ROUGE-2 and RoBERTa. Although the inclusion of semantic data difference signals does not consistently enhance comment generation in semantically rich scenarios, several values are very close to those of the the base versions in several measures. In fact, a closer look to the results show that among the multimodal configurations, the best-performing variant was *NoDiff-Omitted*, which excluded ⟨no_diff⟩ examples during pretraining and focused solely on code that exhibited semantic changes, which achieved the best results in ROUGE-1 and METEOR. The remaining models presented a significant drop compared to the *Code-Only* model. *AST-Graph* achieved intermediate results across most metrics, outperforming multimodal models on most metrics without surpassing *Code-Only*. Notable on all metrics where *NoDiff-Omitted* surpass *Code-Only* it does the same to *AST-Graph*. These results suggests that, although GraphCodeBERT provides richer syntactic and data-flow structure, such structural priors alone appear insufficient to capture the semantics of data-centric transformations. This suggests that pretraining the semantic modality in isolation can capture structural patterns in data transformation but the modality is still too noisy and limits the capacity of multimodal models. This is obvious when analyzing the distribution of human evaluation in Fig. 5, where clearly the introduction of the novel modality resulted in an higher number of samples resulting in a 'poor' score of the generated comments.

The fine-grained coding summarized in Fig. 6 shows that, when meaningful data transformations are present, multimodal variants exhibit slightly lower scores in semantic correctness and completeness compared to the code-only baseline, while maintaining similar levels of syntactic correctness and relevance. This pattern aligns with the automatic metrics reported in Table 5 and suggests that, under our limited data conditions, the semantic modality may introduce some ambiguity that makes it more difficult for the model to consistently associate the symbolic descriptors with the actual transformation, thereby leading to a reduction in comment quality.

Table 5 Evaluation metrics on the Data-Diff Present subset. Bold indicates the best score for each metric across all models

Metrics / Model	BLEU	R-1	R-2	R-L	METEOR	RoBERTa	CodeBERT	H-Eval
Pre-Trained	0.0	27.4	7.0	24.9	18.5	87.3	79.2	1.77
Code-Only	**14.4**	34.0	**23.1**	34.2	29.6	**88.3**	81.1	**2.26**
AST-Graph	10.8	34.1	13.7	33.0	27.5	87.1	80.0	2.19
NoDiff-Omitted	9.2	**38.3**	14.4	**36.4**	**31.6**	88.0	**81.8**	2.10
20% NoDiff	6.6	32.9	13.1	30.3	23.8	87.3	80.1	1.81
Masked-Data Boost	7.0	36.3	17.4	32.6	28.6	87.5	80.0	1.81

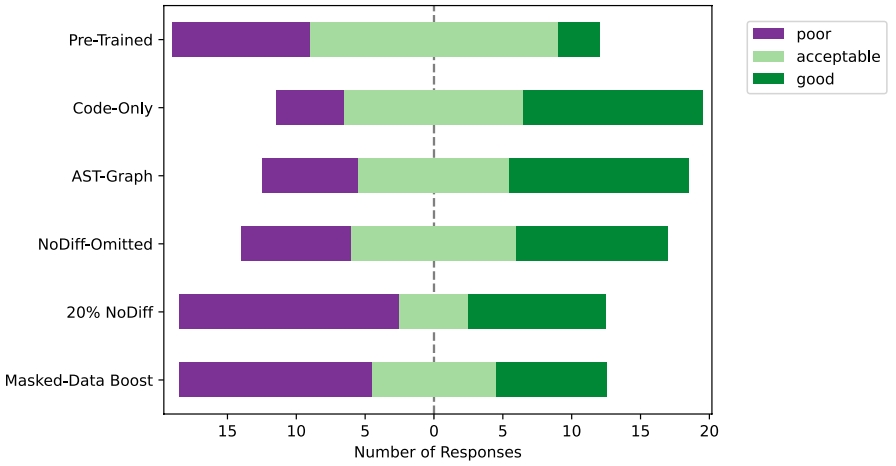

Fig. 5 Human evaluation on the Data-Diff Present subsets

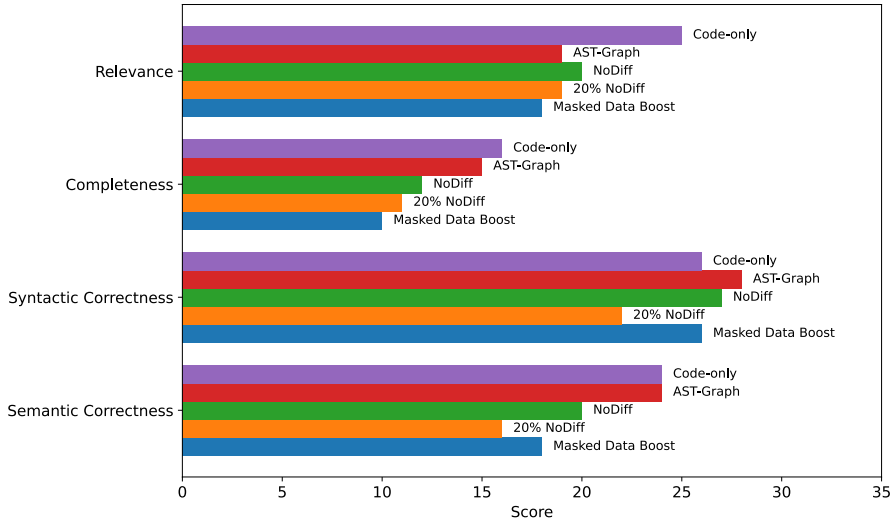

Fig. 6 Qualitative analysis on the Data-Diff Present subsets

6.2 RQ2: robustness in the absence of semantic data differences

In the Data-Diff Absent subset (Table 6), the models incorporating the semantic modality, that in this case all correspond solely to the ⟨no_diff⟩ token, outperformed the code-only baselines. The *Masked-Data Boost* variant model, which emphasized the learning of data difference sequences during pretraining, achieved top scores across BLEU, ROUGE-L and METEOR. *AST-Graph* performed comparably to *Code-Only*.

Table 6 Evaluation metrics on the Data-Diff Absent subset. Bold indicates the best score for each metric across all models

Model \ Metrics	BLEU	R-1	R-2	R-L	METEOR	RoBERTa	CodeBERT	H-Eval
Pre-Trained	0.0	26.4	5.4	22.7	18.1	86.6	78.3	1.45
Code-Only	7.4	46.3	18.1	43.9	33.4	89.2	83.1	2.41
AST-Graph	12.0	45.3	17.9	43.7	32.8	89.2	83.5	2.45
NoDiff-Omitted	8.0	42.2	22.7	39.8	32.2	89.4	81.7	2.36
20% NoDiff	12.2	**49.9**	24.4	46.1	38.9	**90.3**	**83.9**	**2.55**
Masked-Data Boost	**15.3**	49.7	**28.1**	**46.9**	**40.0**	89.3	83.3	2.50

Moreover, the performance gap between the unimodal models and the best multimodal variants grows more pronounced in this setting, with all multimodal models outperforming or matching the baseline. As shown in Fig. 7, while the number of samples scored as 'good' remained comparable, the rate of 'poor' comments was diminished by all multimodal models. This resulted in the multimodal model having the highest average human evaluation score.

As illustrated in Fig. 8, the qualitative coding indicates that *20% NoDiff* and *Masked-Data Boost* generally achieve higher syntactic correctness and relevance, and show more stable semantic correctness, compared to the code-only baseline. These observations align with the quantitative results in Table 6 and support the interpretation that the symbolic ⟨no_diff⟩ token acts as a simple and reliable conditioning signal. We conjecture that the explicit presence of the ⟨no_diff⟩ signal during training acts as a regularizing component that calibrates the multimodal representations and encourages a more conservative generation strategy, thereby reducing unnecessary or speculative details. More specifically, it explicitly indicates that execution did not induce a state change, which constrains the generator and discourages it from attributing transformations that did not occur; this mechanism naturally manifests as gains in semantic correctness by reducing semantic hallucinations. This qualitative

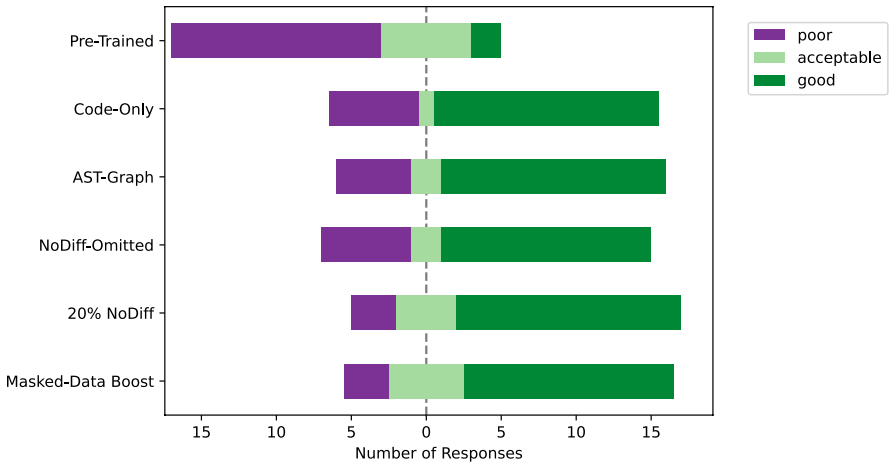

Fig. 7 Human evaluation on the Data-Diff Absent subsets

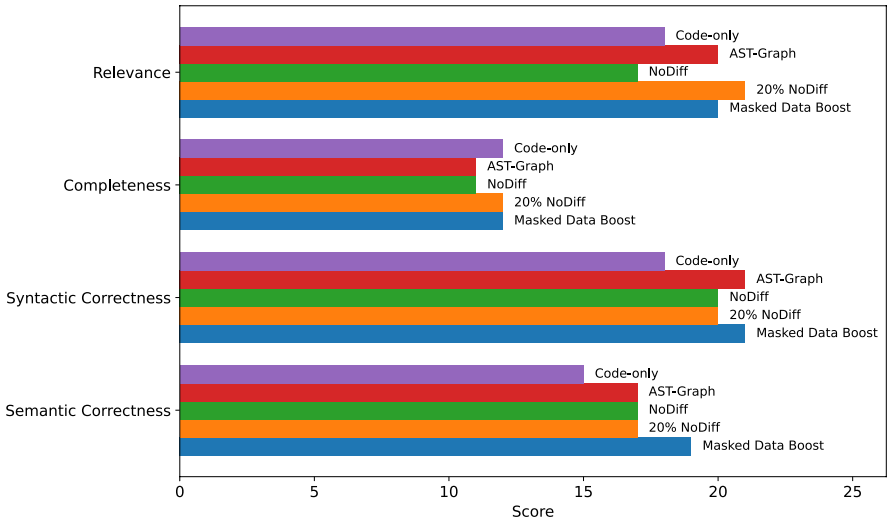

Fig. 8 Qualitative analysis on the Data-Diff Absent subsets

trend complements the improvements reported by automatic metrics and explains why multimodal variants perform particularly well in semantically neutral cases.

7 Discussion

7.1 Interpreting the role of semantic data differences

The results presented in Section 6 offer a nuanced view of the utility of the proposed modality. In scenarios where meaningful data transformations are present (RQ1), the multimodal models did not outperform the code-only baseline. The best results were obtained by the *Code-Only* model, with the closest multimodal variant being the configuration that excluded $\langle \text{no_diff} \rangle$ samples during pretraining. This suggests that while the semantic modality is learnable, the current setup may not fully realize its potential because the amount of semantically annotated training data is insufficient for robust multimodal learning. The comparison with *AST-Graph* further clarifies the complementary roles of structural and execution-derived information. While the graph-based baseline benefits from its richer syntactic and data-flow pretraining, its performance plateaued below that of the best code-only and multimodal variants in both subsets. This highlights that structural priors alone do not fully capture the semantics of data-centric manipulations, reinforcing the relevance of execution-aware signals for documenting data-transformation code.

Importantly, our training set included approximately 25,000 samples, of which only half contained semantically annotated data difference information. This amount is small when compared to the datasets used to pretrain the underlying encoder; for instance, CodeBERT is pretrained on the CodeSearchNet corpus, which contains roughly 6.4 million functions. Although reasonable for a controlled study, this scale

is arguably insufficient for training robust multimodal representations, particularly in the presence of co-attention mechanisms that require large and diverse input distributions to generalize effectively. The limited coverage of transformation types and the reliance on symbolic descriptors may have further constrained the expressiveness of the learned semantic signals. Additionally, data-diff sequence lengths exhibit a heavy-tailed distribution (Table 2), with a few unusually long traces; future work should explore robust handling of these outliers (e.g., normalization or truncation) to further improve generalization.

Nevertheless, even in this constrained setting, the successful integration of structured runtime feedback into a generative architecture serves as a proof of feasibility. The proposed modality was able to encode meaningful patterns, and the model with data encoder, pretrained on data difference alone, demonstrated comparable, and in some cases competitive, performance to strong code-only baselines. These findings indicate that semantic transformation signals, although currently underutilized, are compatible with deep learning-based code summarization frameworks and warrant further exploration. Results from the extended test set (Appendix A) contextualize these observations. Although the absolute performance of all models decreases on more weakly aligned comments, the relative ordering among code-only and multimodal variants remains largely stable in semantically rich cases: the *Code-Only* model continues to lead, while multimodal variants remain competitive despite increased noise. In semantically neutral cases, however, the advantage observed for multimodal models on the curated test set is attenuated, with code-only configurations performing similarly or slightly better. These results provide a more nuanced picture of the approach under noisier supervision: although some gains observed on the curated test set (notably for RQ2) become less consistent at scale, the semantic modality does not appear to introduce fragility and remains broadly competitive. The cross-validation analysis in Appendix B further confirms the stability of these effects across training splits, strengthening the case for semantic execution signals as a promising, though still underexploited, dimension of data-centric code understanding.

The results of Appendix A suggests that the current representation of data differences is not yet precise or expressive enough to guide the model toward the correct transformation semantics. Instead, the symbolic descriptors may introduce ambiguity that the model is not able to resolve with the available data. For instance, many data wrangling operations that alter data content without modifying the schema are represented uniformly as ⟨modified⟩ df, and the modality does not always distinguish between filtering, aggregation, casting, or value-level transformations. Consequently, in semantically rich scenarios, the additional signal may lack sufficient specificity to guide the model toward the correct transformation intent. Conversely, in the Data-Diff Absent subset, multimodal variants improve syntactic correctness and relevance, as shown in Fig. 8, with 20% NoDiff attaining the highest relevance score. Since these cases reduce to the single ⟨no_diff⟩ token, this gain cannot be attributed to additional transformation information at inference time. A plausible explanation is that the data-diff modality acts as a regularizing signal during training, encouraging more conservative and focused generation.

This interpretation is further reinforced by the ablation study reported in Appendix C. When we augmented the semantic representation with explicit value-level transitions, performance did not improve consistently and, in several configurations, degraded slightly. This suggests that simply increasing the amount of runtime detail is insufficient and may introduce additional noise if not supported by substantially larger training data or more expressive modeling strategies.

These findings emphasize that improving the granularity, coverage, and noise robustness of the semantic descriptors is a critical next step for enabling multimodal models to outperform code-only baselines in semantically rich contexts. This limitation is particularly relevant for row-level transformations (e.g., filtering), which are currently captured only as generic modifications based on column-level data changes. Future work should also investigate the impact of training data size and model capacity on the effectiveness of execution-aware signals.

7.2 Unexpected gains in semantically neutral contexts

Perhaps most notably, our findings for RQ2 revealed that in the absence of any semantic transformation (i.e., with the ⟨no_diff⟩ token), multimodal models consistently outperformed the unimodal baselines on the curated test set. The *20% NoDiff* variant model achieved the highest scores in both human evaluation and almost all automatic metrics in the Data-Diff Absent subset. Qualitative results support this interpretation: in the absence of transformations, multimodal variants systematically improved syntactic correctness and relevance, and produced fewer hallucinations or misleading additions. However, this advantage is not fully confirmed by the larger automatic evaluations reported in Appendix A. On the Data-Diff Absent subset of the extended evaluation (Table 9), Code-Only attains higher BLEU and ROUGE-1 scores, while multimodal variants remain competitive but do not systematically surpass the baseline. A similar pattern is observed in the cross-validation analysis (Table 11), where Code-Only retains a slight edge across most metrics. Taken together, these results indicate that the improvement observed on the curated test set does not generalize robustly to broader and noisier evaluation settings.

These results suggest that the inclusion of a symbolic modality, in particular representing no data difference detected during execution, can contribute positively to the generation process in controlled settings with well-aligned reference comments. We hypothesize that the ⟨no_diff⟩ token may act as an informative signal in its own right, encouraging the model to attend more carefully to syntactic features or to adopt a more conservative generative strategy. Alternatively, the multimodal pretraining process may promote more robust and regularized representations, allowing the model to better generalize from limited or ambiguous inputs.

These observations suggest that execution-sensitive architectures may have the potential to stabilize generation in semantically neutral contexts, but that this effect is sensitive to data quality and alignment. At the current stage, the evidence for RQ2 should therefore be considered indicative rather than conclusive. Further investigation on larger and better-aligned datasets is required to determine whether symbolic ⟨no_diff⟩ conditioning yields consistent advantages at scale.

7.3 Toward execution-aware data-centric code understanding

The results presented in this study demonstrate the feasibility of enriching automated code understanding with execution-derived semantic feedback. While the current implementation has focused on comment generation in Python notebooks, the approach suggests broader implications for execution-aware modeling in data-centric software.

Execution-Informed Semantic Representation Learning: Traditional code representation learning has largely centered on static program features: lexical tokens, abstract syntax trees, or data flow graphs. The introduction of structured semantic data differences offers an opportunity to move toward execution-informed embeddings of code. While the results reported in the appendix show that such signals do not yet consistently improve performance in semantically rich cases, they indicate that the models remain relatively stable and do not degrade substantially in semantically neutral contexts. This suggests that execution-aware signals are compatible with transformer-based architectures, but may require richer training data and more expressive representations to fully realize their potential. Future work could therefore focus on improving the alignment between modalities with pretraining strategies better suited to cross-modal learning.

Toward Data-Centric Software Understanding: The symbolic modality introduced in this work, semantic data differences, can serve as a foundation for broader tasks in data-centric software engineering. Future work could investigate whether similar signals support other data-centric tasks, such as detecting unintended data changes during code evolution or summarizing multi-step transformations in notebook workflows. These applications would build directly on the principle demonstrated here: modeling what code does to data, not only how it is written.

Toward Language and Platform Generalization: Although our current approach has been prototyped in the context of Python, the conceptual framework is language-agnostic, as it relies on observing runtime state changes rather than language-specific syntax. The core idea, modeling the semantic effect of code on runtime data, can be generalized across programming paradigms and platforms. Moreover, this modality could support the integration of execution-aware tooling into heterogeneous data science ecosystems. However, such generalization depends on the availability of execution tracing mechanisms and sufficiently consistent data abstractions in the target environment. Our results should therefore be interpreted as a proof of concept within data-centric Python workflows, rather than evidence of immediate cross-language applicability. Extending the approach to other ecosystems remains an open empirical question.

Designing Higher-Resolution Grammars and Hierarchical Abstractions: The current formal grammar for representing data differences is deliberately compact and symbolic. However, richer tasks may require more expressive representations that move beyond atomic operations to reflect higher-level semantic abstractions. A possible direction involves the design of hierarchical semantic grammars capable of encoding transformations at multiple abstraction layers. For instance, low-level changes such as `<modified> df <added_col> log_x` could be composed into mid-level operations like “log-transform”, which in turn map to higher-level intents such as

"feature normalization". This would allow the model to capture both fine-grained operations and higher-level transformation intent within a unified representation.

Connecting with Broader Challenges of reproducibility: Finally, the proposed modality aligns with urgent challenges in scientific reproducibility and research software engineering. In computational research, it is often insufficient to preserve the code alone; one must also capture the transformations it performs, the context in which it is run, and the effects it produces. Automation of code documentation with semantic data differences could serve as lightweight, structured execution logs to enhance reproducibility auditing, complementing existing reproducibility practices by providing a compact, machine-readable summary of data transformations alongside the code. Furthermore, integrating execution-aware comment generation into research pipelines could improve the readability and interpretability of code, a key step toward FAIR-compliant and trustworthy research workflows (Wilkinson et al. 2016).

8 Threats to validity

In this section, we discuss the potential threats to the validity of our findings, organized according to established categories.

Internal Validity: our execution and tracking pipeline relies on a custom instrumentation layer within a controlled virtual environment to capture runtime data changes. While this ensures consistency, any deviation from the original execution environment (e.g. due to unresolved dependencies or stochastic code) may affect the observed data state and thus the extracted data differences. Furthermore, our data tracker compares in-memory variables before and after code cell execution; although effective in most cases, it may miss transformations or operations on non-tracked objects, potentially introducing incomplete semantic information.

Construct Validity: the definition of data differences is central to our approach. However, these are derived through rule-based abstraction over structural changes, which may not fully reflect semantic intent, particularly in complex or domain-specific transformations. Although this strategy ensures robustness and reproducibility, it does not fully capture the semantic diversity of data-wrangling operations. Additionally, row-level transformations are detected indirectly through column-level comparisons and represented as generic modifications. While this design keeps the representation simple, it also introduces noise and limits our ability to distinguish among different transformation types. Moreover, ground truth comments were extracted automatically from inline comments and markdown cells without manual curation. Long comments were optionally summarized using a pretrained model, potentially introducing paraphrasing artifacts. These steps, while practical, may result in training signals that do not always align with the specific transformation being annotated, thereby affecting model alignment and evaluation fidelity.

External Validity: the dataset used in this study is exclusively sourced from publicly available Kaggle notebooks. Although we selected notebooks from a broad range of competition types and domains, this population is inherently biased toward exploratory data science workflows with a competitive or demonstrative nature. Such

notebooks may differ significantly from industrial data-centric software, educational materials, or scientific software used in reproducible research settings. To partially address this concern, our evaluation includes a small held-out test set composed of notebooks shared by researchers. We also expanded our evaluation using an extended test set comprising 2,819 additional samples collected through the same execution pipeline (Appendix A). This larger set is less curated and contains noisier comments, providing a broader, but also less controlled, validation environment. Furthermore, our 5-fold cross-validation (Appendix B) demonstrates that model behavior is stable across different training–validation partitions, reducing the likelihood that our conclusions depend on accidental characteristics of a single training split. Furthermore, our analysis focused primarily on tabular data, which represents a specific and relatively structured data type. As a result, the generalizability of our findings to other data modalities, such as text, images, or time series, is unclear. Broader validation across notebook platforms, data types and user groups remains a necessary step for establishing generalizability.

Conclusion Validity: our evaluation relies on a combination of automatic metrics (e.g., BLEU, METEOR, ROUGE-L, CodeBERTScore) and a human assessment conducted by all three authors. While automatic metrics offer baseline comparability, their ability to reflect semantic adequacy and informativeness in short, context-sensitive comments is limited. To complement this, we performed a manual evaluation in which all authors independently assessed the generated outputs, followed by a joint discussion to resolve disagreements and consolidate scores. We measured inter-rater agreement to quantify annotation consistency, which further strengthens the validity of our qualitative assessment. Furthermore, since both the similarity metric (CodeBERTScore) and the generation model are based on CodeBERT, there remains a risk of evaluation bias due to shared architectural or representational features. Additionally, we complemented automatic metrics and holistic human scoring with a fine-grained qualitative analysis based on four dimensions of documentation quality (semantic correctness, syntactic correctness, completeness, and relevance). This richer qualitative layer provides a more nuanced interpretation and mitigates some limitations of automatic metrics. However, while this triangulation increases confidence in the findings, it does not fully eliminate the risks associated with evaluator subjectivity. As such, our conclusions about the effectiveness of the proposed modality should be considered suggestive rather than conclusive.

9 Conclusions

In this work, we envisioned and introduced a novel modality, semantic data differences, to augment code comment generation by explicitly modeling the post-execution transformations that code induces on structured data. By developing a compact formal grammar for expressing such changes and integrating them into a multimodal architecture via co-attention mechanisms, we proposed a fundamentally new approach to linking code with its runtime effects. This execution-aware perspective, from only considering 'how the code is written' to include 'what the code does to data', suggests new research directions for execution-aware automated software engineering

and for improving the understandability and reusability of data-centric code artifacts. For research software, it is a new direction to improve its reproducibility.

Our empirical evaluation, though based on a newly constructed and openly shared dataset, yielded promising and nuanced results. While the inclusion of data difference information did not consistently outperform unimodal baselines in semantically rich settings (RQ1), it showed benefits in scenarios lacking observable data transformations (RQ2) on the curated test set, although this advantage was not uniformly reproduced in the larger extended evaluation. Qualitative analysis further supported this pattern, showing that the semantic modality helped reduce unnecessary or speculative details when no meaningful transformations were present. These findings were further strengthened by results from the extended test set and cross-validation, which confirmed the overall stability of these trends across different data partitions and evaluation conditions, while indicating that further refinement of the signal and larger-scale training data are needed for it to translate reliably to noisier, in-the-wild samples. Moreover, an explicit ablation study on runtime value-augmented representations (Appendix C) showed that increasing the granularity of semantic descriptors does not automatically yield improvements, highlighting the importance of representation design and data scale in execution-aware modeling. Our comparison with GraphCodeBERT further indicates that structural and data-flow priors alone cannot capture the semantics of data transformations, reinforcing the value of execution-derived modalities. These preliminary results highlight the potential of symbolic runtime signals not only to enrich model representations but also to stabilize and improve generation in ambiguous contexts.

Looking forward, we believe this work sets the stage for deeper explorations of execution-informed modeling in software engineering. Expanding the dataset, refining semantic descriptors and leveraging richer multimodal interactions may lead to systems capable of understanding both code structure and its effects with greater precision. As code increasingly operates within data-driven, exploratory workflows, tools that can leverage what code does and not just how it looks will become essential.

Appendix A: Automatic evaluation on an extended test set

To complement the controlled evaluation performed on the curated test set, we further assessed all model variants on a substantially larger held-out collection of notebook cells obtained through a second execution of the data extraction pipeline described in Section 4.3. This extended test set was produced using the same retrieval, environment reconstruction, and semantic data difference extraction procedures adopted for the main dataset, thereby ensuring methodological consistency, while remaining fully disjoint from the training and validation splits by relying on a new set of notebooks for execution. The resulting dataset comprises 2,819 samples, of which 1,615 contain non-trivial semantic data difference sequences and 1,204 include only the ⟨no_diff⟩ token (indicating an absence of detectable state changes). This partition mirrors the structure of the curated test set and enables population-level validation under more diverse and less controlled conditions.

Unlike the manually curated main test set, where all examples were manually inspected and uninformative comments were filtered out, the extended test set underwent only a minimal filtering. Extremely short, placeholder, or non-linguistic comments were removed; however, no manual screening was applied to address semantic misalignment between reference comments and code. This procedure improves the reliability of automatic metrics but does not fully eliminate misalignments between code and comments. As illustrated by representative examples reported in Table 7, some cells contain comments that are only loosely related to the code they accompany. Consequently, the absolute values of the automatic metrics reported below should be interpreted with caution, as the limited alignment between reference and generated comments affects semantic-oriented metrics.

Table 7 A subset of comments (truncated) we have found unrelated to the soure code

ID	Diff	Code	Comment
404	Yes	`df.head()`	BMI (Body Mass Index) Distribution - Observations : - The BMI values appear to be ...
1175	Yes	`median_value=df['Python'].median()`	Step 2 : Replacement Strategy - Option 1 → Replace with mean ...
1357	Yes	`np.random.seed(42)` `n_samples=100`	The goal is to find a function f that maps input features X to target variable y ...
624	No	`df.head()`	Age : Student's age possibly affecting academic performance ...
606	No	`df.nunique()`	Insights - Students are spending nearly 5 hours ...

Evaluation was conducted using the same automatic measures employed in the main study, reported in Section 5.4, allowing direct comparability with results presented in Section 6. Tables 8 and 9 summarize the results for the Data-Diff Present and Data-Diff Absent subsets respectively. For cases with meaningful semantic transformations, the *Code-Only* and *AST-Graph* baselines continue to achieve the strongest overall performance, in line with the trends observed in the curated test set. Conversely, in cells where execution does not produce observable changes, the multimodal variants perform on par with the unimodal baseline, with the *Masked-Data Boost* configuration showing the most competitive behavior. Notably, when comparing the extended test set to the curated one, we observe that BLEU scores systematically increase across all models, while METEOR, RoBERTa, and CodeBERTScore tend to decrease (especially in the Data-Diff Absent subset). This divergence reflects the nature of the comments in the extended data pool, which are often syntactically simpler or more repetitive (thus inflating n-gram overlap) but semantically weaker or less well aligned with the code, thereby reducing performance on embedding-based measures. The reduced separation between unimodal and multimodal configurations should therefore be interpreted in light of this increased label noise.

Table 8 Evaluation metrics on the Data-Diff Present subset

Model \ Metrics	BLEU	R-1	R-2	R-L	METEOR	RoBERTa	CodeBERT
Pre-Trained	1.2	19.3	4.9	17.9	12.7	84.8	76.0
Code-Only	20.7	33.4	19.2	32.2	26.3	87.3	80.6
AST-Graph	20.7	34.0	19.3	32.6	26.8	87.3	80.9
NoDiff-Omitted	19.3	28.2	15.8	27.3	21.6	86.7	79.2
20% NoDiff	19.7	28.6	16.1	27.3	22.2	86.5	79.2
Masked-Data Boost	19.8	27.8	15.9	26.8	21.6	86.1	78.4

Table 9 Evaluation metrics on the Data-Diff Absent subset

Model \ Metrics	BLEU	R-1	R-2	R-L	METEOR	RoBERTa	CodeBERT
Pre-Trained	0.9	15.8	4.0	14.6	11.0	84.5	74.4
Code-Only	18.4	31.6	18.9	30.6	25.3	86.9	79.3
AST-Graph	15.7	29.8	17.3	28.9	23.4	86.8	79.5
NoDiff-Omitted	14.2	26.8	15.0	25.9	20.2	86.3	78.6
20% NoDiff	13.6	27.1	16.2	26.5	21.1	86.8	79.0
Masked-Data Boost	16.4	27.4	15.7	26.8	21.3	86.6	78.4

Overall, the extended automatic evaluation provides additional evidence that the proposed semantic modality does not introduce instability or degradation in performance, even when reference comments are weakly aligned with code. Rather, model rankings remain consistent, supporting the reliability of the patterns identified in the primary evaluation.

Appendix B: Cross-validation on the new test set

To assess the robustness of the training procedure and mitigate potential sensitivities to a single train–validation split, we performed a 5-fold cross-validation experiment using the training data described in Section 4.3. We adopt five folds for two main reasons. First, 5-fold cross-validation is a widely used convention in empirical software engineering and machine learning, providing a well-established balance between statistical reliability and computational cost. Second, selecting K = 5 preserves a training/validation ratio that closely mirrors the original 80/20 split employed in the main experiment, thereby ensuring continuity with our initial evaluation setting while still enabling a rigorous assessment of model stability across multiple partitions. In each fold, four partitions were used for training and one for validation, while evaluation was consistently performed on the fixed extended test set introduced in Appendix A, as the curated test set is small and less suitable for producing stable variance estimates. This design enables us to isolate variability attributable to model initialization and data partitioning, while preserving comparability across folds.

For each model variant, we computed the mean and standard deviation across folds for all metrics defined in Section 5.4. The pooled results for the Data-Diff Present and Data-Diff Absent subsets are reported in Tables 10 and 11. Across all metrics

and models, standard deviations remain consistently small relative to the corresponding means, indicating stable training behavior. This stability is observed not only for the baselines, but also for all three multimodal variants.

Table 10 5-fold cross-validation results on the Data-Diff present subset

Metrics \ Models	Code-Only	AST-Graph	NoDiff-Omitted	20% NoDiff	MaskedData Boost
BLEU	20.63 ± 0.16	20.23 ± 0.55	19.36 ± 0.50	19.61 ± 0.17	19.33 ± 0.64
ROUGE-1	33.35 ± 1.12	32.75 ± 0.90	28.13 ± 0.76	28.33 ± 0.46	28.42 ± 0.99
ROUGE-2	19.16 ± 0.65	18.65 ± 0.69	16.04 ± 0.40	16.13 ± 0.20	15.96 ± 0.64
ROUGE-L	32.02 ± 1.00	31.43 ± 0.91	27.13 ± 0.65	27.25 ± 0.43	27.40 ± 1.01
METEOR	26.16 ± 0.87	25.30 ± 1.19	21.73 ± 0.78	21.93 ± 0.23	21.86 ± 0.75
RoBerta	87.16 ± 0.17	87.07 ± 0.21	86.51 ± 0.14	86.47 ± 0.16	86.56 ± 0.26
CodeBERT	80.44 ± 0.28	80.43 ± 0.36	79.13 ± 0.17	79.14 ± 0.14	79.11 ± 0.52

Table 11 5-fold cross-validation results on the Data-Diff absent subset

Metrics \ Models	Code-Only	AST-Graph	NoDiff-Omitted	20% NoDiff	MaskedData Boost
BLEU	17.10 ± 1.29	15.22 ± 0.77	15.00 ± 1.73	15.63 ± 1.50	15.08 ± 1.20
ROUGE-1	30.31 ± 1.10	28.77 ± 0.95	26.72 ± 1.06	27.40 ± 0.70	26.39 ± 1.14
ROUGE-2	17.84 ± 1.00	16.67 ± 1.00	15.40 ± 1.06	15.92 ± 0.72	15.25 ± 0.83
ROUGE-L	29.31 ± 1.13	27.84 ± 0.95	26.03 ± 1.05	26.69 ± 0.77	25.74 ± 1.11
METEOR	23.94 ± 1.28	22.20 ± 1.28	20.37 ± 1.20	21.48 ± 0.99	20.32 ± 0.87
RoBerta	86.70 ± 0.14	86.53 ± 0.27	86.41 ± 0.32	86.58 ± 0.22	86.40 ± 0.18
CodeBERT	79.25 ± 0.15	79.04 ± 0.34	78.58 ± 0.25	78.78 ± 0.37	78.50 ± 0.30

Moreover, the relative ranking of models is preserved across folds: *Code-Only* systematically achieves the highest performance in the Data-Diff Present subset, while multimodal variants attain comparable performance to the baseline in the Data-Diff Absent subset. These findings confirm that model behavior does not depend critically on the specific training split, and that the performance observed in the evaluation are reproducible under varied training conditions. The cross-validation analysis therefore strengthens the empirical validity of the trends reported in Section 6 and supports the reliability of conclusions drawn regarding the role and limitations of the proposed semantic modality.

Appendix C: Runtime value-augmented data differences

To further investigate the contribution and design of the proposed semantic modality, we conducted an additional ablation study focusing on the inclusion of lightweight runtime value information within the data difference representation. In the main experiments, as described in Section 3, semantic data differences were encoded using a compact symbolic grammar that abstracts structural changes (e.g., ⟨added⟩,

⟨modified⟩) without incorporating explicit value-level transitions. In this ablation, we extend the original grammar with an additional token ⟨val⟩ to encode simplified runtime value transitions for modified scalar variables. Concretely, when a variable update is detected and both its previous and new values are representable in a compact textual form (e.g., numeric scalars or short categorical values), we augment the corresponding data difference sequence with a value descriptor of the form:

<added> x <val> 5

In all other respects, the dataset construction pipeline described in Section 4.3 is preserved. Using this extended representation, we retrained the multimodal architecture from scratch under the same training protocol described in Section 4.2. All hyperparameters, optimization settings, and masking strategies were kept identical, ensuring that any observed differences are attributable solely to the inclusion of value-level runtime information. The train and validation splits and the curated test set remain unchanged to ensure comparability with results reported in Section 6. We did not perform human evaluation, since these results were produced as an initial check prior to the main experiment and were intended only to probe whether value-level transitions merit further investigation. Moreover, we omit Code-Only and Pre-Trained baselines because the ablation modifies exclusively the data-difference modality and therefore does not affect models that do not consume this input.

As presented in Table 12, results indicate that augmenting the semantic data-difference representation with explicit ⟨val⟩ transitions does not yield consistent improvements over the original symbolic-only formulation. In the Data-Diff Present subset, performance variations are mixed and largely marginal. For the NoDiff-Omitted configuration, the inclusion of ⟨val⟩ tokens results in a decrease in most syntax-based metrics and embedding-based metrics. A similar pattern is observed for 20% NoDiff, where ROUGE-2 improves and ROUGE-L slightly increases, but BLEU, METEOR, and embedding-based scores decline. The Masked-Data Boost variant exhibits small gains in METEOR and ROUGE-L, while BLEU and embedding-based measures remain comparable or decrease slightly. Overall, improvements, when present, are isolated to specific n-gram metrics and are not reflected consistently across semantic similarity measures.

Table 12 Evaluation metrics on the subsets with val (first diff then no diff)

Metrics / Model	BLEU	R-1	R-2	R-L	METEOR	RoBERTa	CodeBERT
Data-Diff Present							
NoDiff-Omitted	9.0	30.8	16.8	31.1	26.3	87.1	79.3
20% NoDiff	5.2	33.3	18.2	32.3	25.4	86.2	78.4
Masked-Data Boost	5.2	35.0	16.4	34.1	29.8	87.2	80.0
Data-Diff Absent							
NoDiff-Omitted	5.5	41.2	17.4	38.1	29.9	88.3	81.7
20% NoDiff	14.2	48.5	22.4	44.9	37.4	89.6	82.8
Masked-Data Boost	9.2	41.5	19.6	39.7	32.9	88.9	81.7

In the Data-Diff Absent subset, where the semantic descriptor reduces to ⟨no_diff⟩, the introduction of runtime information generally results in performance degradation or instability. The NoDiff-Omitted variant shows consistent decreases across BLEU, ROUGE-2, METEOR, and RoBERTa similarity. For 20% NoDiff, BLEU increases, but this gain is accompanied by reductions in ROUGE-1, ROUGE-2, ROUGE-L, METEOR, and embedding-based scores. The Masked-Data Boost configuration experiences a more pronounced drop across nearly all metrics, including BLEU and ROUGE-2, while embedding similarity remains approximately stable.

Taken together, these results suggest that the inclusion of lightweight value-level transitions alters the profile of the semantic modality without providing a consistent benefit for comment generation. While certain configurations exhibit localized improvements in higher-order n-gram overlap (e.g., ROUGE-2), these gains are not systematically supported by METEOR or embedding-based metrics, which more directly capture semantic alignment. In semantically neutral cases, the additional variability introduced appears to reduce stability rather than enhance conditioning.

Overall, the ablation confirms that increasing the granularity of runtime feedback through explicit value transitions does not consistently translate into higher-quality automatic summaries under the current dataset scale and modeling configuration. These findings further support the design choice adopted in the main study to favor a compact, structure-oriented representation of semantic data differences.

Acknowledgements This publication is part of the project PNRR-NGEU which has received funding from the MUR – DM 629/2024, and in collaboration with the Center for Open Science Studies at Politecnico di Torino [4].

Author Contributions Contributor Role Taxonomy (CRediT) https://credit.niso.org/ Conceptualization : All authors; Data curation : GF; Formal analysis : GF; Funding acquisition : FC; Investigation : GF; Methodology : GF, AV, MT; Resources : AV; Software : GF; Supervision : AV, MT, FC; Visualization : GF; Writing – original draft : GF, AV; Writing – review & editing : GF, AV, MT

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement.

Data Availability The datasets and code supporting the findings of this study are available in the Zenodo repository (DOI:10.5281/zenodo.15985078).

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

[4] https://www.polito.it/en/social-impact/polito-libraries/open-science

References

Bansal, A., Haque, S., McMillan, C.: Project-Level Encoding for Neural Source Code Summarization of Subroutines (2021). https://doi.org/10.48550/arXiv.2103.11599.

Bird, C., Ford, D., Zimmermann, T., et al.: Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. Queue **20**(6), 35–5 (2023). https://doi.org/10.1145/3582083

Chen, W., Chen, H.: Collaborative co-attention network for session-based recommendation. Mathematics **9**(12), (2021). https://doi.org/10.3390/math9121392. https://www.mdpi.com/2227-7390/9/12/1392

Şimşek, T., Gülşeni, C., Olcay, G.A.: The future of software development with genai: Evolving roles of software personas. IEEE Eng. Manag. Rev. 1–8 (2024). https://doi.org/10.1109/EMR.2024.3454112

Davila, N., Melegati, J., Wiese, I.: Tales from the trenches: Expectations and challenges from practice for code review in the generative ai era. IEEE Softw. **41**(6), 38–45 (2024). https://doi.org/10.1109/MS.2024.3428439

Dhruv, A., Dubey, A.: Leveraging large language models for code translation and software development in scientific computing. In: Proceedings of the Platform for Advanced Scientific Computing Conference. Association for Computing Machinery, New York, NY, USA, PASC '25, pp. 1–9 (2025). https://doi.org/10.1145/3732775.3733572

Dimitrov, M., Zhou, H.: Unified architectural support for soft-error protection or software bug detection. In: 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), pp. 73–82 (2007). https://doi.org/10.1109/PACT.2007.4336201

Ding, Y., Steenhoek, B., Pei, K., et al.: Traced: Execution-aware pre-training for source code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Association for Computing Machinery, New York, NY, USA, ICSE '24 (2024). https://doi.org/10.1145/3597503.3608140

Donvir, A., Sharma, G.: Ethical challenges and frameworks in ai-driven software development and testing. In: 2025 IEEE 15th Annual Computing and Communication Workshop and Conference (CCWC), pp 00569–00576 (2025). https://doi.org/10.1109/CCWC62904.2025.10903892

Dvivedi, S.S., Vijay, V., Pujari, S.L.R., et al.: A comparative analysis of large language models for code documentation generation. In: Proceedings of the 1st ACM International Conference on AI-Powered Software. Association for Computing Machinery, New York, NY, USA, AIware 2024, pp. 65–73 (2024). https://doi.org/10.1145/3664646.3664765, https://doi.org/10.1145/3664646.3664765

Feng, Z., Guo, D., Tang, D., et al.: CodeBERT: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2020. Association for Computational Linguistics, Online, pp. 1536–1547 (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139. https://aclanthology.org/2020.findings-emnlp.139/

Guo, D., Ren, S., Lu, S., et al.: Graphcodebert: Pre-training code representations with data flow. In: International Conference on Learning Representations (2021). https://openreview.net/forum?id=jLoC4ez43PZ

Haque, S., LeClair, A., Wu, L., et al.: Improved Automatic Summarization of Subroutines via Attention to File Context. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 300–310 (2020). https://doi.org/10.1145/3379597.3387449. https://arxiv.org/abs/2004.04881

Hayes, A.F., Krippendorff, K.: Answering the call for a standard reliability measure for coding data. Commun. Methods Meas. (2007)

Hu, X., Xia, X., Lo, D., et al.: Practitioners' expectations on automated code comment generation. In: Proceedings of the 44th International Conference on Software Engineering. ACM, pp. 1693–1705 (2022). https://doi.org/10.1145/3510003.3510152. https://dl.acm.org/doi/10.1145/3510003.3510152

Huang, J., Guo, D., Wang, C., et al.: Contextualized Data-Wrangling Code Generation in Computational Notebooks. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 1282–1294 (2024). https://doi.org/10.1145/3691620.3695503. https://dl.acm.org/doi/10.1145/3691620.3695503

Jackson, V., Vasilescu, B., Russo, D., et al.: The impact of generative ai on creativity in software development: A research agenda. ACM Trans. Softw. Eng. Methodol. 34(5) (2025). https://doi.org/10.1145/3708523

Kretzer, F., Kolthoff, K., Bartelt, C., et al.: Closing the loop between user stories and gui prototypes: An llm-based assistant for cross-functional integration in software development. In: Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery, New York, NY, USA, CHI '25 (2025). https://doi.org/10.1145/3706598.3713932

Kudriavtseva, A., Hotak, N.A., Gadyatskaya, O.: My code is less secure with gen ai: Surveying developers' perceptions of the impact of code generation tools on security. In: Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing. Association for Computing Machinery, New York, NY, USA, SAC '25, pp. 1637–164 (2025). https://doi.org/10.1145/3672608.3707778

Liu, M., Zhou, F., Chen, K., et al.: Co-attention networks based on aspect and context for aspect-level sentiment analysis. Knowl.-Based Syst. **217**, 10681 (2021). https://doi.org/10.1016/j.knosys.2021.106810, https://www.sciencedirect.com/science/article/pii/S0950705121000733

Lu, J., Batra, D., Parikh, D., et al.: Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. In: Wallach, H., Larochelle, H., Beygelzimer, A., et al. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019). https://proceedings.neurips.cc/paper_files/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

Lu, S., Guo, D., Ren, S., et al.: Codexglue: A machine learning benchmark dataset for code understanding and generation. CoRR arxiv:2102.04664 (2021)

Meem, F.N., Johnson, B.: Investigating the impact of ai-assisted tools on software practitioner well-being. In: Adjunct Proceedings of the 4th Annual Symposium on Human-Computer Interaction for Work. Association for Computing Machinery, New York, NY, USA, CHIWORK '25 Adjunct (2025). https://doi.org/10.1145/3707640.3731915

Mondal, T., Barnett, S., Lal, A., et al.: Cell2Doc: ML Pipeline for Generating Documentation in Computational Notebooks. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 384–396 (2023). https://doi.org/10.1109/ASE56229.2023.00200. https://ieeexplore.ieee.org/document/10298542/

Ni, A., Allamanis, M., Cohan, A., et al.: Next: teaching large language models to reason about code execution. In: Proceedings of the 41st International Conference on Machine Learning. JMLR.org, ICML'24 (2024)

Ozkaya, I.: Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Softw. **40**(3), 4–8 (2023). https://doi.org/10.1109/MS.2023.3248401

Patterson, E., Baldini, I., Mojsilovic, A., et al.: Teaching machines to understand data science code by semantic enrichment of dataflow graphs. arxiv:1807.05691 (2019)

Roy, D., Fakhoury, S., Arnaoudova, V.: Reassessing automatic evaluation metrics for code summarization tasks. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 1105–1116 (2021). https://doi.org/10.1145/3468264.3468588. https://dl.acm.org/doi/10.1145/3468264.3468588

Sanh, V., Debut, L., Chaumond, J., et al.: Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. CoRR arxiv:1910.01108 (2019)

Schröder M, Krüger, F., Spors, S.: Reproducible Research is more than Publishing Research Artefacts: A Systematic Analysis of Jupyter Notebooks from Research Articles (2019).https://doi.org/10.48550/ARXIV.1905.00092.

Spacy: Spacy library (2017). https://spacy.io/

Stalnaker, T., Wintersgill, N., Chaparro, O., et al.: Developer perspectives on licensing and copyright issues arising from generative ai for software development. ACM Trans Softw Eng Method (2025). https://doi.org/10.1145/3743133, just Accepted

Sutton, C., Hobson, T., Geddes, J., et al.: Data diff: Interpretable, executable summaries of changes in distributions for data wrangling. In: Proceedings of the 24th acm sigkdd international conference on knowledge discovery & data mining, pp. 2279–2288 (2018)

Treshcheva, E., Itkin, I., Yavorskiy, R., et al.: Test2text: Ai-based mapping between autogenerated tests and atomic requirements. In: 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 17–20 (2025). https://doi.org/10.1109/ICSTW64639.2025.10962519

Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, NIPS'17, pp. 6000–6010 (2017)

Wilkinson, M.D., Dumontier, M., family=Aalbersberg gigiven=IJsbrand Jan, et al.: The FAIR Guiding Principles for scientific data management and stewardship. Scientif. Data **3**(1), 16001 (2016). https://doi.org/10.1038/sdata.2016.18. https://www.nature.com/articles/sdata201618

Yang, C., Zhou, S., Guo, J.L., et al.: Subtle Bugs Everywhere: Generating Documentation for Data Wrangling Code. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 304–316 (2021).https://doi.org/10.1109/ASE51524.2021.9678520. https://ieeexplore.ieee.org/document/9678520/

Zhou, W., Wu, J.: Code Comments Generation with Data Flow-Guided Transformer. In: Zhao, X., Yang, S., Wang, X., et al. (eds.) Web Information Systems and Applications, vol. 13579. Springer International Publishing, pp. 168–180 (2022). https://doi.org/10.1007/978-3-031-20309-1_15. https://link.springer.com/10.1007/978-3-031-20309-1_15

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Giacomo Fantino[1] · Antonio Vetro'[1] · Marco Torchiano[1] · Federica Cappelluti[2,3]

✉ Giacomo Fantino
giacomo.fantino@polito.it

Antonio Vetro'
antonio.vetro@polito.it

Marco Torchiano
marco.torchiano@polito.it

Federica Cappelluti
federica.cappelluti@polito.it

[1] Department of Control and Computer Engineering, Politecnico di Torino Corso Duca degli Abruzzi 24, Torino, Italy

[2] Department of Electronics and Telecommunications, Politecnico di Torino Corso Duca degli Abruzzi 24, Torino, Italy

[3] Center for Open Science Studies, Politecnico di Torino Corso Duca degli Abruzzi 24, Torino, Italy