The Newton-Puiseux Algorithm and Triple Points for Plane Curves

Execution-Aware Code Documentation through Semantic Data Differences

Giacomo Fantino*
giacomo.fantino@polito.it
Department of Control and Computer Engineering
Politecnico di Torino
Torino, Italy, IT

Abstract

Automated code documentation has gained increasing importance as AI-assisted software engineering tools become integrated into the software lifecycle. However, most approaches rely solely on static features of code, overlooking its dynamic behavior—particularly in data-centric programming environments where code meaning is deeply tied to its effect on data. This research explores how execution-aware code documentation can be achieved by introducing semantic data differences, a structured symbolic representation of how code execution transforms data. By capturing these runtime semantics, this work aims to extend code understanding beyond syntax and lexical features. I propose a multimodal modeling framework that combines static code analysis with post-execution semantics. In support of this framework, I design a data collection pipeline that executes real-world data-centric notebooks, logs variable-level changes, and abstracts them into a grammar of semantic data differences for training and evaluation. This ongoing doctoral research aims to enable documentation systems that describe not only how code is written, but also what code does to data—a step toward more transparent, reproducible, and intelligent data-centric software.

CCS Concepts

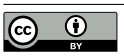
• **Software and its engineering** → **Software maintenance tools**; • **Computing methodologies** → *Supervised learning*; *Natural language generation.*

Keywords

Automated Comment Generation, Machine learning, Multimodal Learning, Runtime Code Analysis

ACM Reference Format:
Giacomo Fantino. 2026. Execution-Aware Code Documentation through Semantic Data Differences. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3774748.3787660

*Completed year 1 out of expected 3. Advisor: Antonio Vetro'; Co-supervisors: Marco Torchiano, Federica Cappelluti.



ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787660

1 Introduction

The automation of software development has rapidly evolved with the advent of generative AI, now capable of generating, refactoring, and documenting code [2, 8]. Despite these advances, these techniques remain largely syntax-bound, relying on surface representations such as abstract syntax trees or token sequences [12]. While these models can describe what code looks like, they struggle to explain what code does—especially in data-centric environments where execution effects determine meaning.

This limitation is critical in contexts like computational notebooks, where data transformations (filtering, reshaping, feature extraction) are central to the program's intent [7]. A single statement may alter a dataset's structure or semantics without leaving clear syntactic cues. For both software engineers and data scientists, understanding such transformations is essential for reproducibility and collaboration, yet documentation in these environments remains sparse and low-quality [5]. This limitation extends beyond productivity: in research software, inadequate documentation hinders reusability and reproducibility, impacting adherence to FAIR principles that underpin sustainable research [9, 10].

My research investigates how execution-derived semantics can improve the contextual understanding of code for documentation purposes. By explicitly modeling runtime data transformations, I aim to enable execution-aware comment generation, bridging the gap between how code is written and what it accomplishes. To learn these behaviors effectively, my research explores training strategies that integrate execution signals directly into the model's representation space. I am currently pursuing the following contributions:

- Defining a structured representation of data differences as a source of semantic context for code summarization.
- Curating a novel dataset of executed Python notebooks enriched with tracked data transformations and corresponding comments, to be openly released.
- Implementing a prototype encoder that integrates both code and data-difference signals.
- Conducting an empirical evaluation of the generated comments, using both automatic metrics and human judgments to assess their quality and informativeness.

2 Proposed Approach

The core idea of this work is to enhance code understanding through semantic data differences: structured, symbolic descriptions of how code execution changes data structures. These differences abstract away raw values and instead encode high-level operations observed during execution, such as additions, removals, or modifications of variables and dataframe columns. A formal grammar

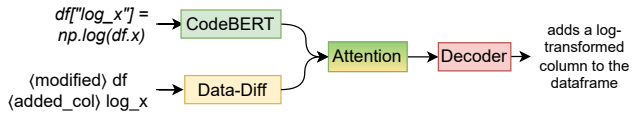


Figure 1: Model architecture

defines these operations, producing interpretable descriptors such as: `<modified> df <added_col> log_x <added> model`. This sequence captures that a dataframe was modified by adding a new column, and a model object was initialized. For source code instances that did not produce a detectable semantic data difference, we assign a dedicated neutral token ⟨no_diff⟩.

The proposed architecture integrates these semantic descriptors with source code using a multimodal encoder-decoder model, as depicted in Figure 1. The architecture combines a CodeBERT-style encoder for static source representations [3] with a transformer encoder specialized for the symbolic data-difference grammar. Code tokens are processed alongside data-difference tokens through a co-attention mechanism, enabling the model to learn interactions between both representations [6]. This allows the system to generate documentation that reflects both structure and behavior.

To operationalize these semantics, the research introduces a novel two-stage training procedure that learns to align code and execution signals, treating execution effects as a first-class symbolic modality within the model. The code encoder is initialized from CodeBERT, already pre-trained for code summarization, and extended with a second encoder for data-difference sequences plus co-attention layers. In a first stage, these two encoders are jointly pretrained with a masked language modeling objective applied over both code and data-diff tokens, so that each modality learns to reconstruct masked tokens while attending to the other. I then explore three variants of this pretraining: **NoDiff-Omitted**, where all ⟨no_diff⟩ samples are excluded; **20% NoDiff**, where both encoders are trained jointly while limiting ⟨no diff⟩ cases to 20% of the pretraining data; and **Masked-Data Boost**, which doubles the masking rate on data-diff tokens to emphasize execution semantics. In the final stage, the fused encoders are paired with the CodeBERT decoder and fine-tuned for the comment generation task.

While the approach is currently instantiated in the context of automated comment generation, the broader research goal is to establish a foundation for execution-aware models capable of understanding data-centric code. This paradigm may ultimately support applications such as semantic debugging, dataflow summarization, and reproducibility tracking in scientific software.

3 Dataset Construction and Evaluation Design

A key enabler of this research is a dataset specifically curated to link code, its execution effects, and natural language comments. We collected 4,869 executable Python notebooks from the Kaggle platform. These notebooks are ideal for studying data-centric code because they contain step-by-step transformations and often include inline comments that describe the intent behind each operation.

Each notebook is executed in a controlled environment instrumented to capture variable-level changes, particularly those affecting dataframes. These logs are abstracted into structured data-difference descriptors following the defined grammar. In addition to execution traces, the dataset includes corresponding code cells and existing human-written comments. When comments exceed a certain length, automatic summarization ensures conciseness while retaining semantic relevance.

The resulting dataset, composed of 55k training and 14k validation samples, supports multimodal learning by pairing code and semantic data differences with explanatory text, though about half lacked comments and were excluded from fine-tuning. This resource also enables controlled evaluation of execution-aware models against traditional code-only baselines. Evaluation combines automatic metrics with human assessment, focusing on whether semantic data differences lead to comments that better capture data transformations and intent.

4 Preliminary Results

I conducted an initial empirical study using the multimodal architecture described in Section 2 and the Kaggle-based dataset in Section 3. The goal was to compare an execution-aware model against a code-only baseline, and to analyze their behavior on two subsets of a small, curated held-out test set of real notebooks: cells where execution produced a non-trivial semantic data difference (Data-Diff Present) and cells where no semantic change was detected (Data-Diff Absent). Models were evaluated with standard automatic metrics and a 3-point human rating by three experienced Python users. Results of the experiment can be visualized in Table 1.

In the Data-Diff Present subset, the code-only baseline remains slightly superior: it obtains the best automatic scores and the highest average human rating, while execution-aware variants sometimes degrade. This suggests that the current data-difference representation and training data are not yet strong enough to consistently help on semantically rich transformations.

Conversely, in the Data-Diff Absent subset, execution-aware models systematically outperform the baseline across both automatic metrics and human scores. These gains arise even though the execution modality collapses to a single ⟨no_diff⟩ token. Rather than conveying semantic information, this constant token may function as a stabilizing cross-modal input: during co-attention, it offers an invariant anchor that helps normalize code representations and reduce spurious attention patterns. We hypothesize that the alignment pressure introduced during joint pretraining improves the fused encoder's robustness and consequently improves the decoder's outputs. Investigating this effect is an important direction for future work.

Overall, these preliminary results indicate that execution-derived signals are already useful to enhance robustness and stability in code regions that do not directly transform data, while their potential in semantically rich cases is not yet fully realized, likely due to limited dataset size and residual noise in the extraction of data differences.

Model	BLEU	ROUGE	METEOR	H-Eval
Data-Diff present				
Code-Only	**14.4**	34.0	29.6	**2.23**
NoDiff-Omitted	9.2	**38.3**	**31.6**	2.10
20% NoDiff	6.6	32.9	23.8	1.81
Masked-Data Boost	7.0	36.3	28.6	1.81
Data-Diff Absent				
Code-Only	7.4	46.3	33.4	2.41
NoDiff-Omitted	8.0	42.2	32.2	2.36
20% NoDiff	12.2	**49.9**	38.9	**2.55**
Masked-Data Boost	**15.3**	49.7	**40.0**	2.50

Table 1: Evaluation metrics on the test set

5 Related Work

Research on automatic code documentation has primarily focused on static features of source code. Neural summarization models such as sequence-to-sequence transformers or pretrained code-language models learn from lexical and syntactic patterns but remain blind to runtime semantics [7].

Other approaches have extended the input context to include variable usage or notebook cell dependencies, improving local coherence but not capturing data transformations themselves [1, 4, 13]. Dynamic analysis tools, on the other hand, produce human-readable summaries of data changes for visualization, but their output is not used as modeling input [11].

This work positions itself at the intersection of these directions by using execution-derived semantics not as explanatory output but as a modeling signal. By formalizing runtime effects into a symbolic modality compatible with deep learning architectures, it extends the representational scope of code understanding beyond syntax and static context.

6 Summary and Future Directions

My research explores execution-aware code documentation as a new perspective on automated software understanding. By incorporating semantic data differences—structured signals derived from code execution, my objective is to bridge the gap between static code representation and runtime behavior.

Early results demonstrate that integrating even symbolic runtime signals can improve model robustness and contextual awareness, especially in semantically neutral code regions. Moving forward, the research will expand in three directions:

- Refinement of semantic representations, by exploring domain specific grammars for different data modalities.
- Scaling the dataset and multimodal training, addressing data sparsity and improving cross-modal alignment.
- Extending execution-aware modeling beyond comment generation to tasks such as semantic trace summarization and reproducibility analysis.

Ultimately, this work aims to contribute toward a new generation of AI-assisted development tools that understand both how code is written and what code does, advancing explainability, transparency, and collaboration in data-centric software engineering.

Acknowledgments

This publication is part of the project PNRR-NGEU, which has received funding from the MUR – DM 629/2024, and was carried out in collaboration with the Center for Open Science Studies at Politecnico di Torino[1].

References

[1] Aakash Bansal, Sakib Haque, and Collin McMillan. 2021. Project-Level Encoding for Neural Source Code Summarization of Subroutines. arXiv:2103.11599 [cs] doi:10.48550/arXiv.2103.11599

[2] Tolga Şimşek, Çağlar Gülşeni, and Gokcen Arkali Olcay. 2024. The Future of Software Development With GenAI: Evolving Roles of Software Personas. *IEEE Engineering Management Review* (2024), 1–8. doi:10.1109/EMR.2024.3454112

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[4] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 300–310. doi:10.1145/3379597.3387449

[5] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh Pennsylvania, 2022-05-21). ACM, 1693–1705. doi:10.1145/3510003.3510152

[6] Jiasen Lu, Dhruv Batra, Devi Parikh, and Stefan Lee. 2019. ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[7] Tamal Mondal, Scott Barnett, Akash Lal, and Jyothi Vedurada. 2023. Cell2Doc: ML Pipeline for Generating Documentation in Computational Notebooks. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Luxembourg, Luxembourg, 2023-09-11). IEEE, 384–396. doi:10.1109/ASE56229.2023.00200

[8] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (2023), 4–8. doi:10.1109/MS.2023.3248401

[9] Max Schröder, Frank Krüger, and Sascha Spors. 2019. *Reproducible Research Is More than Publishing Research Artefacts: A Systematic Analysis of Jupyter Notebooks from Research Articles.* doi:10.48550/ARXIV.1905.00092

[10] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, et al. 2016. The FAIR Guiding Principles for Scientific Data Management and Stewardship. *Scientific Data* 3, 1 (2016), 160018. doi:10.1038/sdata.2016.18

[11] Chenyang Yang, Shurui Zhou, Jin L.C. Guo, and Christian Kastner. 2021. Subtle Bugs Everywhere: Generating Documentation for Data Wrangling Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Melbourne, Australia, 2021-11). IEEE, 304–316. doi:10.1109/ASE51524.2021.9678520

[12] Xuejun Zhang, Xia Hou, Xiuming Qiao, and Wenfeng Song. 2024. A Review of Automatic Source Code Summarization. *Empirical Software Engineering* 29, 6 (2024), 162. doi:10.1007/s10664-024-10553-6

[13] Wen Zhou and Junhua Wu. 2022. Code Comments Generation with Data Flow-Guided Transformer. In *Web Information Systems and Applications*, Xiang Zhao, Shiyu Yang, Xin Wang, and Jianxin Li (Eds.). Vol. 13579. Springer International Publishing, 168–180. doi:10.1007/978-3-031-20309-1_15

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