From Execution to Embedding: Enriching Code Representations with Data Difference Signals for Comment Generation

Giacomo Fantino
giacomo.fantino@polito.it
Politecnico di Torino
Torino, Italy

Antonio Vetrò
antonio.vetro@polito.it
Politecnico di Torino
Torino, Italy

Marco Torchiano
marco.torchiano@polito.it
Politecnico di Torino
Torino, Italy

Federica Cappelluti
federica.cappelluti@polito.it
Politecnico di Torino
Torino, Italy

Abstract

Modern automatic comment generation tools often fail to capture data-centric workflow intents, because syntax alone provides weak signals of how code transforms data. We instead capture semantic data differences—symbolic descriptions of post-execution data transformations—and integrate them with code through a dual-encoder architecture. We evaluate this approach on a dataset of executed Python notebooks pairing code, effect sequences, and human comments. When no transformation is detected, the full pipeline outperforms the baseline across both automatic metrics and human evaluation. When transformations are present, the baseline remains competitive, though some simplified variants surpass it on specific metrics. We make available all software and data to encourage replication and further studies in this area. While our experiments focus on comment generation, our core contribution is broader: we introduce execution-aware embeddings and argue for their applicability to a variety of downstream tasks.

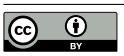
CCS Concepts

• **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, Antonio Vetrò, Marco Torchiano, and Federica Cappelluti. 2026. From Execution to Embedding: Enriching Code Representations with Data Difference Signals for Comment Generation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786826



ICSE-NIER '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2425-1/2026/04
https://doi.org/10.1145/3786582.3786826

1 Introduction

Generative AI is now widely applied across the software engineering lifecycle [3, 13], impacting a range of activities—including requirements, writing, reviewing and testing code [2, 4, 7, 15]. Yet, these tools operate mainly on static code representations—tokens, ASTs, or graphs [17]. However, modern software often performs data transformations whose purpose is not evident from the code's surface syntax alone. This is pronounced in data-centric workflows, where single statements can manipulate data with little lexical evidence [11]. Systems trained only on static views therefore risk brittle or generic interpretations precisely where understanding the effect of code on data would be most valuable [6].

For this reason, we propose a new approach to understanding code, which captures not just how code is written, but also its observable semantic impact on data, thus enabling embeddings that encode *what effects programs have on data*, not just *how code is written*. Therefore, we introduce semantic data differences—a compact representation of the transformations code performs on structured data—formalized into structured descriptors designed for learning. This novel approach integrates code syntax analysis with capturing concrete effects, providing a fundamentally more grounded basis for generating accurate and interpretable comments, better aligned with real data workflows.

In this paper, we also present preliminary evidence supporting the approach. Our pilot study required three key artifacts: first, an extended grammar to represent data effects compactly and interpretably; second, a dataset construction pipeline that instruments and executes Python notebooks, emitting paired (code, effect, comment) triples at the cell level; third, a model architecture that integrates code and effect representations for comment generation. All artifacts are openly released with the scientific community[1] for replications and adaptations of the technique: while comment generation is our first application, our broader vision is an execution-aware foundation for code embeddings that shifts representations from static syntax to dynamic signals of how code transforms data.

In the remainder of the article, we provide an overview of the state of the art and how our proposal differ from it (Sec. 2), then we describe details of the approach and its implementation, respectively in Sec. 3 and Sec. 4. We continue by providing the experiment design (Sec. 5), the results (Sec. 6) and their interpretation (Sec. 7). We conclude providing the research plan ahead (Sec. 8) and a synthesis of what was achieved (Sec. 9).

2 Motivation and Background

Although static models of code excel at capturing syntax and data dependencies, they struggle to capture implicit semantic impact [17]. Data wrangling and pipeline steps are often expressed through operators whose runtime effect is not recoverable from tokens alone [11]. Execution-sensitive representations are a viable remedy: they are grounded in observed changes to program state, yet must be designed to avoid high-entropy raw traces that are unwieldy for learning. Prior work has leveraged dynamic analysis to present execution traces directly to humans, mainly as documentation aids for notebooks and pipelines [14, 16]. Beyond these efforts, recent approaches have begun to expose LLMs to runtime signals to strengthen model understanding of program behavior [1, 12].

Our work takes a different stance: we treat execution signals as machine-learnable representations, not human-readable summaries, enabling alignment with code and applicability beyond comment generation. Unlike LLM-based techniques that feed full traces inside prompts, we introduce a compact grammar that capture only high-level state changes, avoiding noisy, high-entropy traces while remaining expressive enough to support downstream tasks. This distinguishes our approach from both trace-based prompting and prior documentation tools: we align code embeddings with semantically structured data-difference operators, making execution information a first-class representational signal within the embedding space.

3 Proposed Approach

Our approach rests on three elements: a compact execution grammar, an execution encoder trained with grammar-aware objectives, and cross-modal alignment to a code encoder via co-attention.

We model post-execution changes as short token sequences that name variables, data structures, and high-level operations. Concretely, a data difference is computed by snapshotting tracked program objects before and after a cell executes and comparing the resulting states. Any detected creation, removal, or structural modification is mapped to the corresponding operator in our grammar, yielding a compact symbolic description of the state change. The grammar is intentionally minimal to reduce noise and encourage compositionality, while remaining extensible to represent richer effects:

$$
\begin{aligned}
\mathcal{G} \quad ::= \quad & \langle \text{no\_diff} \rangle \\
| \quad & \langle \text{added} \rangle \; v \\
| \quad & \langle \text{removed} \rangle \; v \\
| \quad & \langle \text{modified} \rangle \; v \\
| \quad & \langle \text{modified} \rangle \; df \; \langle \text{added\_col} \rangle \; c \\
| \quad & \langle \text{modified} \rangle \; df \; \langle \text{modified\_col} \rangle \; c \\
| \quad & \mathcal{G} \; \mathcal{G}
\end{aligned}
$$

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, for the statement `df["log_x"] = np.log(df.x)` the tracker emits ⟨`modified`⟩ `df` ⟨`added_col`⟩ `log_x`, representing that the DataFrame `df` was modified by adding the column `log_x`. 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). Hereafter we denote by ⟨op⟩ any operator token from the grammar, and by arg the immediate argument following ⟨op⟩ (e.g., for ⟨`added`⟩ `df`, df represents the argument). The neutral case ⟨no_diff⟩ is treated as a singleton operator with no argument.

The formal grammar we propose draws inspiration from prior work such as WrangleDoc [16] and Data Diff [14], 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 framework, detailed in the following section.

4 Method

4.1 Training set construction

We construct a paired corpus that aligns source code with a symbolic description of its effect on program state. Public Python notebooks are executed in a controlled sandbox that restores declared dependencies, enforces timeouts, and snapshots tracked objects before and after each cell. A lightweight tracker emits a short sequence of data-difference tokens drawn from our grammar; if no semantically relevant state change is detected we emit the special token ⟨no_diff⟩. Inline comments were split into sub-cells to increase supervision density, and long comments were automatically summarized with Spacy following [11]. Each sample thus consists of code, its data-difference sequence, and (if available) the comment or title. Executing 4,869 notebooks yielded 55k training and 14k validation samples, though about half lacked comments and were excluded from fine-tuning. The datasets are released with our artifact[1].

4.2 Model architecture

Our model follows an encoder–decoder design that fuses code with data differences, as shown in Figure 1. The code encoder is initialized from CodeBERT [5], a transformer-based model pretrained on source code and natural language on the CodeXGLUE dataset [10]; the difference encoder is a RoBERTa-style transformer trained from scratch over the difference vocabulary [8]. We fuse code and execution signals through a shallow co-attention layer [9], which allows tokens in one modality to directly attend to those in the other. This design captures cross-modal relevance patterns and conditions a standard transformer decoder with self-attention over the fused encoder embeddings, initialized from CodeXGLUE comment-generation training and later fine-tuned autoregressively

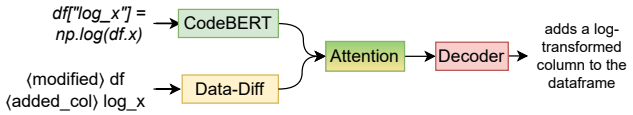


Figure 1: Model architecture

on our dataset. For example, when conditioned on the fused representation of `df["log_x"] = np.log(df.x)` and ⟨`modified`⟩ `df` ⟨`added_col`⟩ `log_x`, the decoder produces a brief cell-level comment such as "adds a log-transformed column to the dataframe." This design keeps the difference encoder usable as an execution-aware representation beyond comment generation, while allowing the decoder to exploit interactions between how code is written and what it does to data. The model is available in the repository[1].

4.3 Training objectives and schedule

Training proceeds in three stages:

Stage I — Data-encoder pretraining. We train the RoBERTa-style difference encoder on our corpus of difference sequences with a tailored curriculum. For the first epochs we exclude ⟨no_diff⟩ cases and optimize a combined masked-LM and edit-reconstruction loss, where the model predicts operator tokens from embeddings. Masking is targeted to reflect the structure of the grammar: in 40% of cases we mask operators, in 30% we shuffle them, and in 30% we perturb arguments (e.g., ⟨added⟩ df → ⟨added⟩ arg). This curriculum emphasizes stable representations of transformation operators while introducing variability in arguments and ordering, promoting robustness and compositional generalization. We then reintroduce ⟨no_diff⟩ with a binary discrimination loss to calibrate neutral cases.

Stage II — Cross-modal pretraining. We compose the difference encoder with the initialized code encoder and the co-attention stack. We optimize a multi-objective loss combining masked modeling over both modalities and a cross-modal operator reconstruction term that predicts masked ⟨op⟩ tokens using only the code encoder representation, encouraging the code embeddings to reflect data-difference semantics.

Stage III — Task fine-tuning. We attach the decoder pretrained on CodeXGLUE and fine-tune the full encoder–decoder model on our dataset, with the decoder performing autoregressive generation while attending over both code and data-difference representations.

5 Experiment Design

To assess whether data differences improve automated comment generation, we conduct an empirical study centered on 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?

We evaluate model variants on two complementary test subsets reflecting the presence or absence of semantic state change. For instances exhibiting a detected transformation, we quantify the contribution of explicit difference signals (RQ1). For instances without a measurable transformation, we examine whether the model maintains robustness and generalization (RQ2). This design enables a controlled comparison across conditions of varying semantic availability while holding evaluation protocols fixed. We report results for four models:

Baseline Code-Only training (Code-Only): modality-free baseline with only code encoder and no data-diff encoder or fusion.
Without no-diff calibration (NoDiff_Cal): it skips the second part of Phase I, i.e., it does not introduce ⟨no_diff⟩ and does not train the binary no_diff discriminator; this isolates the value of neutral-case calibration in the diff encoder.
Without cross edit reconstruction (EditRecon): it removes the Phase II edit reconstruction term, i.e., no cross-modal ⟨op⟩ reconstruction from embeddings, to test if co-attention still passes meaningful operator information without the explicit pressure.
Full pipeline with co-attention (Full-CoAttn): it includes all components and objectives, acting as the primary system.

Automatic evaluation uses BLEU, ROUGE, and METEOR to assess surface quality, and we complement them with embedding-based metrics similarity, RoBERTa and CodeBERTscore, to assess deeper semantic alignment. Human evaluation was conducted by three annotators: two co-authors and one independent evaluator. Each annotator judged on a 3-point Likert scale based on usefulness and semantic correctness: 3 for accurate, clear, and insightful; 2 for partially correct or vague but still helpful; and 1 for misleading, broken, or generic. All conflicts in evaluations were discussed and resolved in a dedicated reconciliation meeting. To answer RQ1, we analyze the subset with non-neutral differences and test whether access to data differences improves both automatic and human scores. To answer RQ2, we analyze ⟨no_diff⟩ samples and test whether the model maintains or improves performance relative to Code-Only.

All experiments ran on a single consumer GPU; full details and dependencies are listed in the reproducibility package [1].

6 Results

In this section, we report the outcomes of our empirical evaluation. In Table 1 we present both automatic evaluation metrics and results from human annotation, following the experimental design described in Section 5.

RQ1: Impact of Semantic Modality When Data Transformations Are Present

Across automatic metrics on the subset with detected transformations, Code-Only remains the overall best performer. Interestingly, the NoDiff_Cal variant surpasses both EditRecon and Full-CoAttn across several metrics and even outperforms Code-Only on ROUGE-1 and METEOR. Human evaluation paints a slightly different picture: Code-Only attains the highest average usefulness score overall, but within the data-diff variants, Full-CoAttn is judged marginally more useful than NoDiff_Cal or EditRecon.

RQ2: Robustness in the Absence of Semantic Data Differences

When no measurable transformation is observed the picture inverts: the full model substantially improves over Code-Only on every reported automatic metric, more substantially for BLEU and ROUGE-L, with corresponding gains in encoder similarity scores. Human evaluation confirms this pattern: EditRecon and Full-CoAttn are rated clearly higher than Code-Only, indicating that annotators also perceived these variants as more semantically correct and useful.

[1]Link to source code and data to replicate the experiment

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	METEOR	RoBERTa	CodeBERT	Human-average
Data-Diff present								
Code-Only	**14.4**	34.0	**23.1**	34.2	29.6	**88.3**	**81.1**	**2.23**
NoDiff_Cal	8.4	**37.6**	20.7	**36.9**	**32.6**	87.7	80.4	2.06
EditRecon	8.6	26.2	12.1	25.5	25.0	87.0	78.6	2.06
Full-CoAttn	7.0	29.2	9.0	28.2	21.5	87.0	78.7	2.10
Data-Diff Absent								
Code-Only	7.4	46.3	18.1	43.9	33.4	89.2	83.1	2.41
NoDiff_Cal	8.8	39.4	21.5	38.3	31.0	89.0	81.3	2.36
EditRecon	9.6	42.2	21.6	40.5	33.7	89.2	82.3	2.45
Full-CoAttn	**15.5**	**53.4**	**33.7**	**52.0**	**43.5**	**90.5**	**84.6**	**2.64**

Table 1: Evaluation metrics on the test set

7 Discussion

The results presented in Section 6 provide a nuanced perspective on the promise and current limitations of the proposed modality. When meaningful transformations are present, the Code-Only baseline remains competitive overall; however, the NoDiff_Cal variant surpasses it on specific automatic metrics. Human judgments confirm that Code-Only attains the highest overall scores, while within the data-diff variants, Full-CoAttn is rated marginally higher than the other configurations. In contrast, when no meaningful data difference is detected, the full pipeline clearly outperforms Code-Only across both automatic metrics and human evaluation. We attribute the weaker performance on transformation-heavy samples primarily to data scarcity and noise in present-diff sequences, which limit the model's ability to consistently learn fine-grained operator semantics. In contrast, we hypothesize that the neutral ⟨no_diff⟩ signal provides a stable semantic anchor that regularizes the co-attentional encoder, reducing hallucinations and improving comment quality when no state change occurs. Overall, these findings highlight an important insight: execution-aware embeddings already provide tangible robustness benefits in neutral contexts, even before they fully capture fine-grained transformations. This positions them as an immediately useful representational modality, while also pinpointing a clear next step for research: scaling data so that transformation signals are more effectively harnessed. Taken together, the results answer RQ2 positively, execution-aware representations improve robustness without explicit changes, while showing for RQ1 that the approach is promising but not yet sufficient. This clarifies where future scale, modeling depth, and cleaner data are most likely to pay off.

8 Future Plans

We aim to mature execution-aware code embeddings into a general, empirically grounded methodology. Near term, we will harden the data pipeline and broaden coverage beyond Kaggle by executing a more diverse notebook corpus. Concretely, we will extend instrumentation to track effects on additional object types and side-effects, and we will reduce label noise by refining comment extraction.

On representation, we will evolve the symbolic execution grammar in two practical directions. First, by incorporating a small set of higher-resolution operators to capture common composite actions (for example, grouped aggregations, joins, and model-fit/predict cycles) while preserving compactness. Second, we will explore hierarchical abstractions learned from traces so that low-level edits compose into mid-level operations and intent-level descriptors, enabling more stable learning signals.

To demonstrate generality beyond comment generation, we will evaluate on retrieval and change-impact tasks that explicitly test whether embeddings capture 'what code does to data,' and we will retain ⟨no_diff⟩ scenarios to probe robustness in neutral contexts; metrics will combine automatic measures with targeted human judgments of semantic adequacy.

9 Conclusions

We introduced an execution-aware modality based on compact data-difference signals and demonstrated its integration into a dual-encoder architecture. Our evaluation shows that the full pipeline consistently improves over Code-Only baselines when no data transformations are present, while it is surpassed by its simplified versions in presence of transformation-heavy samples. These preliminary results highlight a clear opportunity: execution-aware embeddings already provide tangible benefits in neutral contexts, while pointing toward future progress through richer and cleaner transformation data. Although we demonstrated the approach on comment generation, the underlying modality is general and naturally extends to other software engineering tasks (trace-based summarization, execution-aligned search, or semantic anomaly detection). We therefore view this line of work as an execution-aware representation path that is immediately useful today and promising for capturing deeper semantic effects as the approach scales.

10 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[2]. The authors also thank Lorenzo Laudadio, PhD candidate at Politecnico di Torino, for his contribution to the human evaluation through the assessment of model-generated comments.

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

References

[1] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 220, 13 pages. doi:10.1145/3597503.3639183

[2] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (Jan. 2023), 35–57. doi:10.1145/3582083

[3] 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* 53, 4 (2024), 34–39. doi:10.1109/EMR.2024.3454112

[4] Nicole Davila, Jorge Melegati, and Igor Wiese. 2024. Tales From the Trenches: Expectations and Challenges From Practice for Code Review in the Generative AI Era. *IEEE Softw.* 41, 6 (Nov. 2024), 38–45. doi:10.1109/MS.2024.3428439

[5] 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

[6] 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

[7] Felix Kretzer, Kristian Kolthoff, Christian Bartelt, Simone Paolo Ponzetto, and Alexander Maedche. 2025. 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 (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 879, 19 pages. doi:10.1145/3706598.3713932

[8] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL] https://arxiv.org/abs/1907.11692

[9] 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

[10] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation, In Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track. *CoRR* abs/2102.04664.

[11] 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

[12] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. NEXT: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1540, 28 pages.

[13] 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

[14] Charles Sutton, Timothy Hobson, James Geddes, and Rich Caruana. 2018. 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* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2279–2288. doi:10.1145/3219819.3220057

[15] Elena Treshcheva, Iosif Itkin, Rostislav Yavorskiy, and Nikolai Dorofeev. 2025. Test2Text: AI-Based Mapping between Autogenerated Tests and Atomic Requirements. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 17–20. doi:10.1109/ICSTW64639.2025.10962519

[16] 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

[17] 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