Hybrid Static Analysis and Large Language Models for Detecting Uninitialized Variables in Python

Nadav Eyal
Tel Aviv University
`nadaveyal1@mail.tau.ac.il`

Yaniv Tzipin
Tel Aviv University
`yanivtzipin@mail.tau.ac.il`

Abstract

Identifying undeclared variables—variables used before they are defined—is a critical aspect of static code analysis. While this task is part of a broader space of bug detection, we focus specifically on this subproblem due to its frequent occurrence and subtlety. Traditional static analysis tools like pylint can catch some cases, but they are inherently limited by heuristic rules and lack contextual understanding. In this work, we aim to complement such lightweight static analysis tools using a neural approach. We fine-tune a sequence-to-sequence model (CodeT5) (Wang et al., 2021) to generate undeclared variable names from buggy Python functions. To support this, we construct a synthetic dataset by injecting undeclared variable bugs using AST-based program transformations. We also experiment with different decoding strategies to represent target variables. Our model is evaluated on a held-out test set and compared with pylint. Results show that the model outperforms pylint, achieving higher recall and precision in identifying undeclared variables. These results highlight the potential of language models to augment or enhance traditional static analysis techniques in focused error domains.

1 Introduction

Detecting uninitialized variables is a critical step in static code analysis, helping identify potential runtime errors and logical flaws in software systems. This task is particularly important in dynamically typed languages like Python, where variable declarations are implicit and the risk of referencing undeclared variables is higher. Traditionally, compilers or linters like pylint are used to catch such bugs through predefined rules and static analysis techniques. While effective in many cases, these methods can be restricted and might overlook more subtle or context-dependent bugs. In this work, we investigate an alternative, learning-based approach to the task of identifying undeclared variables. Specifically, we pose the problem as a sequence-to-sequence task and fine-tune a pretrained language model (CodeT5) to predict undeclared variable names given a buggy Python function. Unlike static analyzers, our model can implicitly learn patterns from data and generalize beyond rule-based heuristics. To support this goal, we construct a novel dataset of buggy Python functions where undeclared variable usages are injected in a controlled and diverse manner. In fact, we design a modular pipeline that uses Python's Abstract Syntax Tree (AST) to strategically inject bugs across different code structures — including inside loops, conditions, and scopes — making the task both realistic and challenging. We explored two target representation schemes: character-level decoding and word-level decoding using special delimiters to separate predicted variable names. Our work contributes a hybrid method that blends neural modeling with code-aware preprocessing, and offers a comparative evaluation against baseline tools like pylint. We show that our model can learn to accurately predict undeclared variables and generalize beyond rule-based patterns.

2 Related Work

Early approaches to code defect detection modeled programs as sequences or graphs. For instance, Zhang et al. (2019) introduced ASTNN, which uses Bi-GRU models over sequences of statement trees derived from abstract syntax trees (ASTs) to capture syntactic structures, while Zhou et al. (2019) employed Graph Neural Networks (GNNs) to encode structural information such as data and control flow. While effective in small-scale settings, these methods were trained on limited datasets and primarily framed as coarse-grained classification tasks, limiting their generalization and practical applicability. With the rise of Transformer-based models in natural language processing (Vaswani et al., 2017), similar architectures have been successfully extended to programming languages. Several pretrained models have been proposed for code understanding and defect detection, including CuBERT (Kanade et al., 2020), CodeBERT (Feng et al., 2020), and CodeT5 (Wang et al., 2021). These models have been trained on large-scale source code corpora and adapted to a variety of code tasks, such as summarization, translation, and classification. Notably, CuBERT introduced benchmark datasets for detecting code defects like Variable Misuse, Wrong Binary Operator, and Swapped Operand. However, it was designed for single-label classification and does not support structured or generative outputs, making it ill-suited for tasks like undeclared variable detection, where multiple and symbolic outputs are required.

Bui et al. (2022) proposed CodeT5-DLR, a unified framework built on CodeT5 for jointly addressing function-level bug detection, line-level bug localization, and program repair. By formulating these tasks as complementary objectives and training them together, CodeT5-DLR demonstrated improved performance over single-task models. However, it focuses on general-purpose defects and outputs binary labels, line spans, or fixed code sequences—not symbolic identifiers.

In contrast, our work focuses on a specific and structurally distinct class of semantic errors: undeclared variable usage. We frame this as a sequence-to-sequence generation task, using a fine-tuned CodeT5 model to output a variable-length sequence of undeclared variable names, conditioned on the buggy function. This formulation enables the model to detect not just the presence of a bug, but to infer its symbolic cause—which variables are missing—rather than producing classification labels or repaired code. Unlike CodeT5-DLR, our model is trained in an ad hoc fashion, using a synthetically generated dataset tailored to undeclared variable detection, enabling precise supervision and evaluation. Furthermore, our results show that this approach outperforms static analysis tools like `pylint` in both precision and recall.

While large-scale general-purpose models like ChatGPT (Brown et al., 2020; OpenAI, 2024) demonstrate strong performance on code generation tasks, they lack the domain-specific supervision and task specialization needed for structured classification tasks such as undeclared variable prediction. In this context, CodeT5 offers a practical and performant alternative, especially when fine-tuned for structured outputs aligned with program semantics.

3 Methodology

3.1 Dataset Construction

To train and evaluate our model for undeclared variable detection, we constructed a custom dataset by injecting use-of-undeclared-variables bugs into existing Python functions. As a starting point, we used the Python Functions Filtered dataset from Hugging Face,[1] which contains 58,220 cleaned and syntactically valid Python functions. These functions served as a clean, high-quality corpus for controlled bug injection.

3.2 Code Representation and Metadata

Each function is represented by a lightweight *CodeSnippet* structure, storing both its original and buggy forms, along with minimal metadata (e.g., control-flow features and undeclared variable labels) needed for injection and evaluation.

3.3 Bug Injection Strategy

We implemented multiple bug injection methods, combining static analysis with AST and control-flow analysis. Each injection method preserves syntactic correctness and targets semantic correctness violations related to undeclared variable usage.

3.3.1 Control-Flow-Guided Assignment Removal

We constructed a control-flow graph (CFG) for each function using a custom CFG builder that supports branching constructs such as `if`/`else`, `while`/`for`, and `try`/`except`. From the CFG, we identified assignment statements that define a variable used later along at least one path in the graph. By removing the corresponding assignment node from the AST, we created bugs in which variables are referenced without being declared. This ensures that the injected bug is both realistic and reachable at runtime.

3.3.2 Declaration-in-Condition Injection

To simulate scoping errors, we injected a variable declaration inside the body of a control struc-

[1] https://huggingface.co/datasets/vikp/python_functions_filtered

ture (e.g., within an `if` block) and inserted usages of that variable outside its defining scope. The injected variables are randomly named and initialized with realistic values such as integers, strings, or lists. Their usage is contextually appropriate—for example, passed to `print()`, used in arithmetic, or accessed via indexing. This models a common class of bugs where developers unintentionally use variables outside their lexical scope.

3.3.3 Probabilistic Multi-Bug Injection

Bug injection is applied probabilistically to each snippet. A random choice is made between the deletion and conditional-injection strategies. Each snippet is visited multiple times, which allows for multiple bugs to be injected into a single function. This creates a richer dataset that includes both simple and compound undeclared variable scenarios.

3.4 Final Dataset Format

Each record in the final dataset is a JSON-style row containing key information for evaluation and traceability. Table 1 summarizes the fields in the final dataset.

Table 1: Fields included in the final exported dataset.

Field	Description
`original_code`	The original, unmodified function.
`buggy_code`	The same function after bug injection.
`uninitialized_vars`	A list of variable names used without declaration.
`num_uninitialized_vars`	The number of undeclared variables in the buggy code.

The final dataset is publicly available on Hugging Face[2] and includes both buggy and clean examples to support consistent evaluation.

3.5 Dataset Statistics

Our final dataset consists of 58,220 Python functions, each containing a varying number of undeclared variable bugs, as illustrated in Figure 1.

[2] https://huggingface.co/datasets/Ynaiv/buggy-python-functions

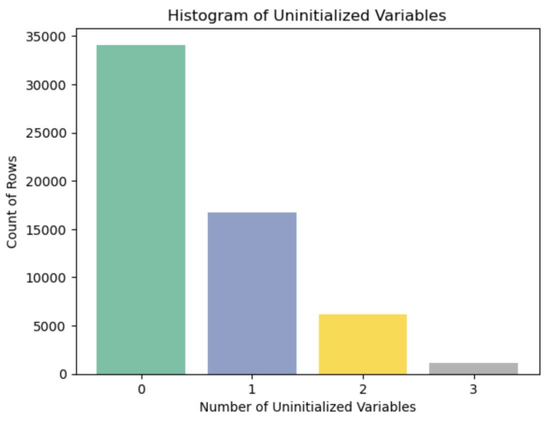

Figure 1: Histogram of uninitialized variable counts across all code snippets.

3.6 Model Training and Learning Phase

Our task involves identifying undeclared variable usage in Python functions, which requires understanding both the structure and semantics of code. To capture these nuances, we framed the problem as a sequence-to-sequence (seq2seq) generation task: given a buggy function as input, the model is trained to output a set of undeclared variable names. This setup allows us to handle a variable-length and unordered output space—something traditional classification models struggle with.

3.6.1 Model Choice

We selected `Salesforce/codet5-base`, a pre-trained encoder-decoder Transformer specifically adapted for code-related tasks. CodeT5 is based on the T5 architecture, which has shown strong performance in generation tasks. Compared to encoder-only models (e.g., CodeBERT), the seq2seq structure of CodeT5 is well-suited to our goal of generating a set of variable names. CodeT5 also incorporates code-aware pretraining objectives, making it more effective at modeling syntax and identifier relationships in source code.

3.6.2 Input/Output Representation

Each input to the model is either a buggy or a valid function, tokenized using `AutoTokenizer` with a maximum input length of 512 tokens. The output is a set of undeclared variable names, joined into a single string and separated by the `|` delimiter. This format is concise and allows the model to generate multiple variable names without enforcing an artificial order. For functions without undeclared variables, we use a special placeholder token `<NO_UNINITIALIZED_VARS>`.

3.6.3 Dataset Splitting

We split the full dataset of 58,220 functions into training (80%), validation (10%), and test (10%) sets using stratified sampling to ensure a balanced distribution of undeclared variable counts. This partitioning supports both model development and final evaluation. The dataset is organized as a `DatasetDict`.

3.6.4 Training Configuration

We fine-tune the model using the `Seq2Seq-Trainer` from Hugging Face Transformers with the following hyperparameters: 3 epochs, batch size 8, learning rate 2×10^{-5}, and weight decay 0.01. We use mixed-precision (FP16) training when supported, and report metrics to TensorBoard for tracking. We adopted the hyperparameters suggested in the CodeT5 paper (Wang et al., 2021), which were shown to be effective for similar generation tasks. The batch size of 8 was selected based on GPU memory constraints.

3.6.5 Evaluation Metric

Our primary evaluation metric is *set accuracy*, which measures whether the set of predicted variable names exactly matches the set of ground truth undeclared variables, disregarding order and duplicates.

4 Results and Discussion

We evaluate the performance of our fine-tuned CodeT5 model against a static analysis baseline — `pylint`. The primary task is to detect undeclared variables in buggy Python functions. Each detector produces a set of variable names, and we compare these against the ground truth using four metrics: Precision, Recall, F1 Score, and Exact Match. We use F1 Score as the primary metric because it balances precision and recall, which is critical in this set prediction setting, where correctly identifying all missing variables without introducing spurious predictions is essential.

4.1 Overall Performance

Table 2 summarizes the results on the test set. Our model outperforms `pylint` across all metrics. Notably, while `pylint` achieves high precision (0.903), it suffers from low recall (0.568), indicating frequent under-reporting. In contrast, our model maintains both high precision (0.988) and high recall (0.968), resulting in an F1 score of 0.978 and Exact Match of 0.979.

Table 2: Overall comparison between `pylint` and our model on the test set.

Detector	**Precision**	**Recall**	**F1 Score**	**Exact Match**
`pylint`	0.903	0.568	0.697	0.811
Model	**0.988**	**0.968**	**0.978**	**0.979**

4.2 Performance by Structural Feature

To further probe generalization, we evaluated both detectors across function subsets containing structural constructs like `if`, `try`, and `while`. As shown in Table 3, our model excels in every feature category. This robustness is especially important for real-world code, which often includes non-trivial control flow.

4.3 Per-Feature Improvement Analysis

To better understand the source of our model's gains, we visualize the per-feature improvement in performance over `pylint` in Figure 2. Each cell represents the difference in Precision, Recall, or F1 Score between the model and `pylint`, averaged over a specific control flow feature.

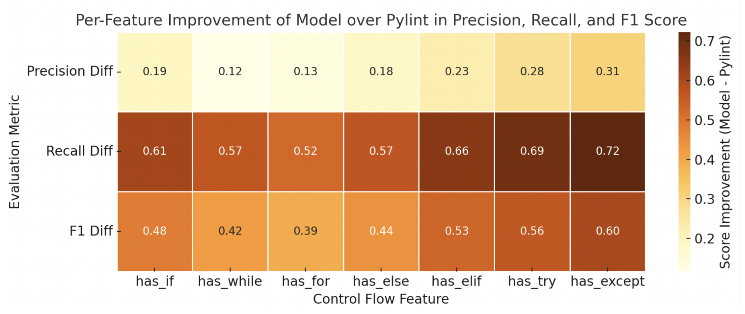

Figure 2: Per-feature improvement of our model over `pylint` in Precision, Recall, and F1 Score. Each cell indicates the score difference (Model - `pylint`).

As shown in the heatmap, the largest recall gains appear in features with more complex control flow, such as `try`, `except`, and `elif`, where the model outperforms `pylint` by over 0.65 points in recall. Precision differences are more modest but still noticeable across all features. This reinforces the conclusion that our model is better equipped to handle semantic

Table 3: Performance comparison across structural features on the test set.

Feature	Detector	Precision	Recall	F1 Score	Exact Match	Num Samples
`has_if`	Pylint	0.798	0.361	0.498	0.513	1959
	Model	**0.987**	**0.973**	**0.980**	**0.968**	1959
`has_while`	Pylint	0.876	0.422	0.570	0.429	198
	Model	**0.992**	**0.988**	**0.990**	**0.975**	198
`has_for`	Pylint	0.858	0.454	0.593	0.700	530
	Model	**0.992**	**0.977**	**0.985**	**0.979**	530
`has_else`	Pylint	0.801	0.402	0.535	0.552	891
	Model	**0.983**	**0.968**	**0.976**	**0.964**	891
`has_elif`	Pylint	0.743	0.307	0.434	0.467	362
	Model	**0.971**	**0.963**	**0.967**	**0.948**	362
`has_try`	Pylint	0.711	0.307	0.429	0.510	198
	Model	**0.989**	**0.994**	**0.992**	**0.989**	198
`has_except`	Pylint	0.677	0.271	0.387	0.440	159
	Model	**0.987**	**0.994**	**0.990**	**0.987**	159

context and variable scoping logic in non-trivial control structures.

4.4 Qualitative Example

To further illustrate the strengths of our hybrid approach, consider the following function from the test set:

```
def avoidhexsingularity(rotation):
   epsilon = 1e-12
   if abs(diagnostic) < epsilon / 2.0:
      rotation_adjusted = rotation +
          epsilon
   else:
      var = ['hello', 60]
      rotation_adjusted = rotation
   str(var)
   return rotation_adjusted
```

The ground truth undeclared variables for this function are `[diagnostic, var]`.

Model prediction: `diagnostic | var`
Pylint prediction: `diagnostic`

This example highlights two types of variable-related bugs from our dataset:

- `diagnostic` was **removed** as part of our *deletion-based injection strategy.*
- `var` was **injected** inside a conditional scope, following our *control-flow-guided injection.*

While pylint correctly identifies `diagnostic`, it fails to detect `var`, which is defined in the `else` branch but accessed later outside its scope. Our model, in contrast, successfully identifies both undeclared variables—demonstrating a better understanding of control flow and scoping semantics.

4.5 Discussion

The empirical results strongly support our approach. While `pylint` provides decent precision in simple cases, it fails to generalize to more complex code structures and often misses undeclared variables. Our fine-tuned CodeT5 model consistently detects both the presence and identity of undeclared variables with high accuracy, regardless of code structure. This demonstrates the advantage of learning-based methods for semantic bug detection and motivates their integration into static analysis pipelines.

5 Conclusion

Our results demonstrate that a fine-tuned CodeT5 model—trained specifically to detect undeclared variables—achieves high precision and recall, complementing a widely used static analysis tool, `pylint`. This performance gain is attributed to the model's ability to learn from the structure and semantics of code, and to its training on a dedicated task-specific dataset. Importantly, the model complements traditional static analysis rather than replacing it. While lightweight tools like `pylint` are fast and interpretable, they often struggle with semantic bugs or complex control flows. Our findings highlight the promise of combining static methods with targeted LLM-based detectors. Looking ahead, we envision an ecosystem of specialized lightweight models, each trained to detect a narrow class of code issues with high precision and recall. Such models can be composed into a hybrid analysis framework—combining algorithmic static rules with learned semantic understanding—to provide deeper and more robust code quality assurance.

References

Allamanis, M., Jackson-Flux, H., and Brockschmidt, M. (2021). Self-supervised bug detection and repair. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ayewah, N., Pugh, W., Morgenthaler, J., Penix, J., and Zhou, Y. (2008). Using static analysis to find bugs. *IEEE Software*, 25(5):22–29.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Bui, D., Phan, L., Nguyen, H., Nguyen, A. T., and Nguyen, T. N. (2022). Improving code understanding and repair via multi-task learning with codet5. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Chen, N., Sun, Q., Wang, J., Gao, M., Li, X., and Li, X. (2023). Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 14401–14414. Association for Computational Linguistics.

Developers, P. (2024). Pylint user manual.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Gonzalez, J. E. (2019). Abstract syntax tree (ast) and control flow graph (cfg) for program analysis. Technical Report.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Cubert: Bert for control flow tasks in source code. *arXiv preprint arXiv:2001.00059*.

Lhoest, Q., Minixhofer, B., Schmid, P., von Platen, P., Patry, F., Rault, T., Gesmundo, A., Debut, L., Sanh, V., and Wolf, T. (2021). Datasets: A community library for natural language processing. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 175–184.

Li, H., Hao, Y., Zhai, Y., and Qian, Z. (2024). Enhancing static analysis for practical bug detection: An llm-integrated approach. In *Proceedings of the ACM on Programming Languages (OOPSLA)*.

Mohialden, Y. K., Mohialden, F. H., and Farhan, H. H. (2023). A comparative analysis of python codeline bug-finding methods. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 14(7):602–610.

OpenAI (2024). Gpt-4 technical report.

Pradel, M. and Sen, K. (2018). Deepbugs: A learning approach to name-based bug detection. In *Proceedings of the ACM on Programming Languages*, volume 2, pages 147:1–147:25.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. https://arxiv.org/abs/1910.10683.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008.

Wang, Y., Kang, S., Mishra, P., Gu, X., and Singh, S. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8697–8708.

Zhang, Z., Wang, Y., and Liu, Y. (2019). A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10197–10207.