RefineID: A Developer-Centric IDE Assistant for Better Identifiers

Eya Jeljli
Dalhousie University
Halifax, Canada
eya.jeljli@dal.ca

Tushar Sharma
Dalhousie University
Halifax, Canada
tushar@dal.ca

***Abstract*—Identifiers constitute a linguistic backbone of source code, directly shaping its readability, maintainability, and long-term comprehension. Yet developers continue to encounter inconsistent, unclear, and suboptimal naming choices that erode code quality over time. Although prior studies have explored identifier naming, existing solutions often operate outside the developer's workflow and do not provide adaptive, in-context support. To address this gap, we present RefineID, a development-oriented assistant that evaluates and refines identifiers directly within the coding environment. Integrated into Visual Studio Code, the tool operates seamlessly in the background, maintains responsiveness, and incorporates user feedback to capture their intent and promote consistent naming practices. By leveraging existing AI models, RefineID demonstrates how readability-oriented analysis and support can be embedded directly into everyday development workflows, improving developer productivity and code maintainability.**

***Index Terms*—Identifier naming, Readability, Technical debt, Software maintenance, Developer tools, Code quality, Refactoring**

I. Introduction

Source code serves as a medium of collaboration and knowledge transfer. Programs are read far more often than they are written, making readability a fundamental aspect of software quality. The quality of identifiers *i.e.,* the names of variables, methods, classes, and other program elements, play a key role defining readability [1], [2]. Identifiers serve as a bridge between human intent and source code while a software developer attempts to read, understand, refactor, or extend the functionality supported by the code. Moreover, identifiers account for up to 70% of a typical software project in terms of number of characters [3], highlighting the scale of their importance.

Despite the importance of well-chosen identifiers, developers continue to encounter inconsistent, unclear, and suboptimal identifiers. In a large empirical study, Beniamini *et al.* [4] found that 36% of all Java identifiers used abbreviations omitting vowels or consonants, and 10% consisted of single characters. Similarly, research on linguistic antipatterns has identified recurring naming inconsistencies, such as identifiers whose names contradict their role in the implementation, which are known to impair code comprehension [5]. Therefore, identifying poor identifiers and suggesting more appropriate alternatives are essential for maintaining the readability and sustainability of software systems.

Despite the importance, current tool support remains limited and fragmented. For example, modern IDEs provide rename refactorings, but they use a developer-supplied name. The IDEs do not, natively, offer any assistance in detecting an unsuitable identifier and proposing a better alternative [6]–[8]. Similarly, static analysis tools such as ESLint, Checkstyle, and SonarQube flag violations of style but restricted at surface-level checks, leaving the burden of checking the suitability of an identifier and coming up with a better one on developers [9]–[13]. From the research side, several studies have explored identifier quality assessment and renaming through linguistic and learning-based techniques to detect linguistic antipatterns [14] and naming inconsistencies [15], [16], and to suggest better identifiers [17]–[20].

Most of these research efforts result in a trained model. However, even if the models are publicly available, they are not integrated with development environment. Furthermore, several large language models (LLMs) are now available within popular IDEs. One may use available LLMs to generate good identifiers using appropriate prompts. However, prompting an LLM every time for each identifier may be cumbersome and degrades developer experience. Also, keeping developers' preferences and history of accepting or rejecting renaming suggestions are not supported by IDE extensions such as Copilot. In summary, though there are several AI models, including generic LLMs and models that are trained specifically for identifier renaming, their seamless integration with developers' tools and environment is missing, inhibiting adoption by the developers.

To address the limitations of existing tools and research, we introduce **RefineID**, a Visual Studio Code extension that automatically evaluates the suitability of code identifiers and suggests improved alternatives using AI models. The extension supports local models as well as remote LLMs accessed through APIs. Locally, it can work with any model trained for identifier renaming, including those served via Ollama.

The *primary goal* of RefineID is to simplify the process of identifying and refining suboptimal identifiers for developers. Specifically, the extension continuously scans the loaded code files in the VS Code IDE in the background without blocking the developer's workflow, detects identifiers using abstract syntax trees, and queries the configured model to assess their contextual suitability. Based on this analysis, it provides alternative suggestions, if required, in a developer-friendly interface, allowing developers to review and optionally apply the proposed changes. RefineID thus acts as a collaborative assistant that enhances developer judgment and prevents the silent accumulation of unsuitable identifiers.

The main **contributions** of RefineID are a) a seamlessly integrated VS Code extension that analyzes identifiers automatically and provides context-aware refinements without disrupting the developer's workflow, b) a backend architecture combining lightweight lexical heuristics with masked language model inference—whether local, through Ollama, or cloud-based—to balance accuracy, privacy, and responsiveness, and c) a mechanism that records developer feedback, preserves historical naming decisions, bridging human judgment and automated intelligence. The source code of the extension and installable VS Code extension (.vsix file) is available online[1]. The VS Code Marketplace release of RefineID can be accessed here[2]. The demonstration video can be accessed here[3].

II. Related Work

Recent studies have explored identifier quality assessment and renaming through both linguistic and learning-based techniques. Peruma *et al.* [14] proposed *IDEAL* to detect linguistic antipatterns, and later introduced a linguistic appraisal model to evaluate and suggest appropriate identifiers [17]. Zhang *et al.* [18] employed a machine-learning framework combining semantic and historical features to predict and recommend identifier renamings. Zhang *et al.* [15] further addressed naming consistency using a code-dictionary method that detects semantic and grammatical inconsistencies based on domain-specific vocabularies. Together, these works demonstrate a clear evolution from rule-based and linguistic analyses to data-driven and context-aware models for improving identifier quality.

Furthermore, studies have employed transformer-based large models to leverage their richer textual understanding in identifying unsuitable identifiers and proposing alternatives. Wang *et al.* [16] benchmarked multiple deep-learning architectures for detecting inconsistent method names, revealing that transformer-based generation models capture semantic mismatches but struggle to generalize beyond balanced datasets. Liu *et al.* [20] introduced a global-context Transformer that synthesizes local, project-wide, and documentation cues to generate coherent method names. Vijayvargiya *et al.* [19] advanced this direction by fine-tuning masked language models through multi-mask objectives to refine identifier naming accuracy, while Mastropaolo *et al.* [21] compared T5 and CuGLM models for automatic variable renaming, showing strong performance yet continued difficulty with multi-word and rare identifiers. Collectively, these approaches showcase the promise of Transformer-based models for identifier reasoning. However, to enable developer adoption, these techniques and the models trained by them need to be integrated into developers' workflow which is precisely the main goal of the proposed tool.

[1]https://github.com/SMART-Dal/RefineID

[2]https://marketplace.visualstudio.com/items?itemName=RefineID-Assistant.refineid

[3]https://www.youtube.com/watch?v=Fn8IY5FQWtw

III. RefineID

A. System Architecture

RefineID is implemented as a Visual Studio Code extension designed for responsiveness and minimal disruption, currently supporting Python and Java. We choose Visual Studio Code given that it is one of the most widely used IDEs, as reported by the PYPL Top IDE Index [22]. Figure 1 outlines the architecture of the tool, highlighting four main components: extension frontend, backend, model providers, and persistence. The frontend of the extension captures events from the IDE and presents a user interface. The frontend extensively uses CodeLens [23], a programmable feature offered by VS Code IDE, to highlight unsuitable identifiers and offer options to the developers. Specifically, the frontend asynchronously submit identifier assessment and renaming requests to the backend and the backend responds to each of the requests when results are available using Server-Side Events (SSEs). The backend, implemented using FastAPI, hosted in a docker container, orchestrates analysis and communicates with a local or remote AI model. The extension also sports a lightweight persistence layer realized by SQLite database to store identifier metadata, generated suggestions, and developer feedback.

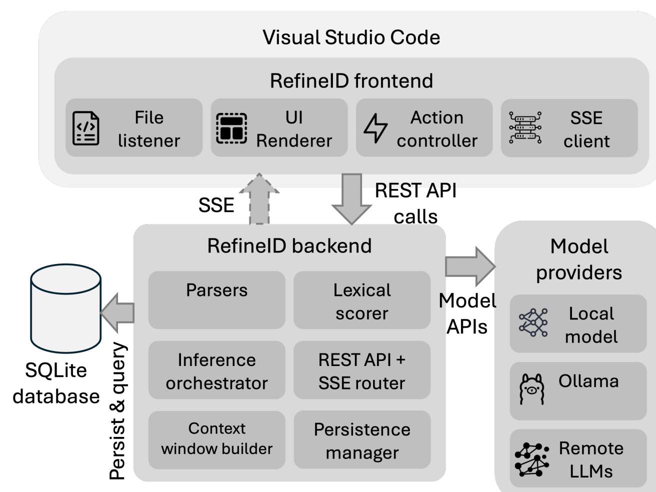

Fig. 1: RefineID System Architecture.

In the rest of the section, we elaborate on the implementation details of the extension.

B. End-to-End Identifier Refinement Process

Step-1 Incremental analysis of source files: When a source file is opened in the IDE, the extension computes its hash and identifies the programming language. The frontend then issues a request to the RefineID backend, sending the file path, language (i.e. Python or Java), and source code to be enqueued for analysis. The backend then queries the database to determine the analysis scope. If the file is known and unchanged, previous results, such as identifier statuses, suggestions, and user decisions, are restored to avoid unnecessary compute. If the file is new or modified, the backend performs a fresh analysis and records any new or updated identifiers.

Step-2 Parsing and occurrence tracking: The backend, based on the language used, parses the code and generates the Abstract Syntax Tree (AST) using the `ast` package for Python, or `javalang` in case of Java. We focus on these two languages given their wide adoption by practitioners [24]. It then extract all identifiers along with their definition and usage locations.

Step-3 Prefiltering: Certain identifiers must not be renamed, for example, framework-specific (*e.g.,* Java field annotated with `@JsonProperty(``user_id'')`), generated code (*e.g.,* files under `target/generated-sources/` and generated `UserDto`), and external identifiers (*e.g.,* identifiers inside dependency folders such as `node_modules/` or `site-packages/`). We label such identifiers as skipped and do not consider them for renaming. All other identifiers are marked as *renamable* and proceed to lexical and semantic analysis.

Step-4 Lexical heuristics: RefineID applies convention-based heuristics to each identifier, considering factors such as length, casing style, and the presence of digits, following standard naming rules such as Python's PEP 8 [25] and Java naming conventions [26]. The result is a lexical verdict (*good* or *bad*), that we store together with an explanatory rationale. We use this verdict later in the process to determine whether to present a suggestion to the developer regarding the identifier in question.

Step-5 Tokenization and context-window strategy: For each identifier set to be renamable, the backend loads the model's tokenizer and constructs masked context windows around its declaration and usages. RefineID groups occurrences by their enclosing class or method, encodes the surrounding block of code, and extracts two window types: a definition-focused window and additional usage windows selected from clustered occurrence locations. Each window is trimmed to fit the model's token limit.

Step-6 Masking and subtoken handling: Within each context window, all occurrences of the target identifier are replaced by the model's mask token. For multi-subtoken identifiers [19], the backend inserts a sequence of consecutive mask tokens of varying length for each evaluation, allowing the model to consider renaming options of varying subtoken lengths rather than constrained by the original identifier's structure. Masking is performed independently for each window and for each subtoken length setting. The backend also records the mask positions so that pseudo log-likelihood (PLL) [27] scores can be computed consistently across all window variants.

Step-7 Candidate generation and ranking:

For each window, the model generates candidates of one to three subtokens, and computes corresponding PLL scores for each candidate within that window. These scores are then aggregated across all windows associated with the same identifier to capture its overall contextual fit. Based on these aggregated scores, RefineID produces up to four suggestions: the globally best-scoring candidate (the winner), regardless of length, and the top candidates within each length bucket (one-, two-, and three-token options).

However, not every renamable identifier is sent to the model on every analysis pass. To avoid redundant computation, respect user intent, and maintain responsiveness, RefineID applies several conditions that bypass inference and presentation of the candidates to the user. Table I summarizes the cases where the backend deliberately avoids invoking the model.

Condition	Effect on Inference
User confirmed or rejected	User confirmation prevents re-analysis of the applied name, while rejection reflects the developer's intent to never receive suggestions for that identifier.
Cached suggestions exist	If a suggestion has already been previously generated and stored, the backend reuses it and skips new model inference unless the file changes.
Skip rules triggered	Framework-bound, code-generated, external/vendor, or reserved identifiers are tagged as *skipped* during prefiltering and are not included in the renamable set.
Lexically good and strong original PLL	If the current identifier is lexically good, and semantically outperforms its candidates within the decision threshold, the backend suppresses presentation and avoids re-running generation until the code context changes.

TABLE I: Cases where RefineID avoids invoking model inference.

Together, these conditions ensure that RefineID requests model inference only when it is genuinely useful, avoiding redundant work, eliminating repeated prompts, and preserving a fast, unobtrusive developer experience.

Step-8 Presentation decision logic: If the lexical verdict (from Step 4) for an identifier is good, the backend also evaluates PLL of the original identifier by unmasking it within the same context window. The PLL values for both the original identifier and the winning candidate are then calibrated against the mean PLL of their respective length buckets. If the decision delta, the calibrated difference between the winner and the original, falls below the configured threshold, the backend discards the suggestion and records the candidate set. Otherwise, the suggestions are presented to the user.

Step-9 Asynchronous presentation and user interaction: RefineID presents identifiers asynchronously by scheduling per-identifier inference jobs through a bounded thread pool, allowing multiple identifiers to be analyzed in parallel. As results become available, the backend streams updates to the IDE via a server-sent events (SSE) channel. The extension listens to this stream and gets the new suggestions for the identifier in question. If there is a new suggestion available for an identifier, the extension places a small RefineID icon in the IDE gutter beside the line where the identifier is defined first. This design keeps the interface lightweight and avoids disrupting the user's workflow with unnecessary visual clutter. When the user hovers over the identifier, the extension reveals, as shown in Figure 2, a contextual CodeLens showing the winning suggestion along with following actionable options.

Fig. 2: RefineID in VS Code

- **Confirm** to use the presented candidate to apply rename refactoring,
- **Reject** to retain the original identifier and record the decision in our database so that it is not suggested again,
- **See more** to present the other suggested options,
- **Hide** to not show any suggestion for the identifier in the current session, and
- **Undo** to revert a previously confirmed rename and restore the original identifier.

C. Model Providers

RefineID decouples its inference engine from the extension through a flexible model provider layer supporting three modes: *(i)* a local ModernBERT model fine-tuned to predict good identifier given a context, *(ii)* locally hosted LLMs via Ollama [28], and *(iii)* remote LLMs accessed via APIs.

For the LLM-based modes, the backend uses a small set of prompt templates that encode naming guidelines, JSON output schema, and code context. These prompts were refined manually during early testing. The exact templates and their invocation logic are available in the replication package in the extension's source code. The cloud provider layer accepts any OpenAI-compatible endpoint and includes built-in integrations for OpenAI, Anthropic, Gemini, and Mistral, as well as aggregators such as Groq and OpenRouter, which allows the extension to target most models exposed through such APIs.

The local model serves as the default model, offering privacy guarantees, while external providers can be enabled as needed.

Local model: We fine-tuned ModernBERT [29], a model supporting masked-modeling suitable for suggesting an effective identifier given a context window with masked identifier. For fine-tuning, we adopt the approach proposed by Vijayvargiya *et al.* [19], with the primary difference being our choice of base model. We select ModernBERT over GraphCodeBERT because it provides a substantially larger context window, which is important for our task. The remainder of our approach follows the original method, including the training procedure and dataset. The fine-tuned model reports training and evaluation losses of 1.5 and 2.5, respectively, and corresponding perplexities of 1.98 and 12.18. The model is encapsulated in the RefineID backend that is download while constructing the docker image. It is important to note that achieving state-of-the-art performance in identifier renaming is not the goal of this work; rather, our objective is to provide a reasonable default model that complements the extension.

IV. Preliminary Evaluation

A. Setup

We select Apache Commons IO[4], a popular Java library maintained by Apache Software Foundation for our preliminary evaluation. The library offers common I/O utility classes. Specifically, we analyzed `org.apache.commons.io.file` package of the selected repository using RefineID. For this experiment, we used a Groq-hosted Llama model `llama-3.3-70b-versatile`. We analyzed all Java files in the target package (excluding its subdirectories), totaling 16 Java files comprising 4 789 lines of code and containing 492 identifiers.

B. Analysis Results

Out of the 492 identifiers, 14 identifiers (2.8%) were skipped, while the remaining cases yielded clear lexical verdicts (i.e. were considered for renaming). The tool classified the majority, 307 identifiers (62.4%), as *good*, whereas 171 identifiers (34.8%) were flagged as *bad*. In response, the tool generated 441 suggestions, including both winning recommendations and alternative candidates, highlighting opportunities to improve identifier quality within the analyzed package.

Although Apache Commons IO is generally regarded as a well-maintained and a high-quality project with over a thousand stars, RefineID identified a notable number of potentially unsuitable identifiers (34.8% of the total). A closer look at the proposed replacements shows that, while many original names are not severely flawed, the suggested alternatives offer clearer semantics and more explicit intent. For instance, RefineID recommends `appendDirectory` in place of the vague `add`, and `hashCodeValue` instead of the less descriptive `result`. We plan to conduct a systematic evaluation not only to evaluate the quality of suggested identifiers but also to assess usability of the proposed tool.

A recurrent pattern among problematic names involved very short identifiers or abbreviations whose meaning was only implicit in surrounding comments or code. In contrast, the suggested names tended to be longer but more descriptive (for instance, replacing a generic helper-style name with one that references the specific file operation being performed). This aligns with RefineID's design objective of prioritizing readability and intent-revealing names rather than enforcing purely stylistic changes.

[4]https://github.com/apache/commons-io

V. Conclusions and Future Work

RefineID revisits how identifier naming can be supported in modern development environments. Instead of adding an external analysis tool, it places naming assistance directly within the developer's regular workflow. Through contextual reasoning and a minimal interface, RefineID promotes clearer naming choices without interrupting focus. Its hybrid design, combining lightweight local processing with optional remote LLM support, shows that readability-oriented assistance can be both practical and responsive while respecting developer autonomy.

In the near future, we plan to expand language support and improve compatibility with a wider range of large language models to accommodate different development settings. We also aim to enhance the local ModernBERT model for stronger contextual understanding and more consistent suggestions. Beyond these steps, our long-term goal is to grow RefineID into a broader refactoring assistant that helps developers reason about and improve code quality across multiple dimensions.

Acknowledgments

The lead author was sponsored by Mitacs, via Globallink Research Internship during the summer of 2025. We are grateful to Anshul Singh for helping us fine-tune the ModernBERT model used in the local inference mode.

References

[1] I. R. da Silva Simões and E. Venson, "Measuring how changes in code readability attributes affect code quality evaluation by large language models," 2025. [Online]. Available: https://arxiv.org/abs/2507.05289

[2] R. Gresta, V. Durelli, and E. Cirilo, "Naming practices in java projects: An empirical study," in *Proceedings of the Brazilian Symposium on Software Quality (SBQS '21)*. ACM, 2021.

[3] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 602–614.

[4] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 45–54.

[5] V. Arnaoudova, M. D. Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 20, no. 4, p. 103–134, 2015.

[6] A. Feldthaus and A. Møller, "Semi-automatic rename refactoring for javascript," in *Proceedings of the 2013 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013, pp. 323–338. [Online]. Available: https://cs.au.dk/~amoeller/papers/renaming/paper.pdf

[7] JetBrains, "Rename refactorings in intellij idea," https://www.jetbrains.com/help/idea/rename-refactorings.html, accessed: 3 Nov 2025.

[8] Microsoft, "Rename refactoring in visual studio," https://learn.microsoft.com/en-us/visualstudio/ide/reference/rename?view=vs-2022, accessed: 3 Nov 2025.

[9] "Eslint rules reference," https://eslint.org/docs/latest/rules/, OpenJS Foundation, 2025, describes ESLint's rule system, including stylistic rules and configuration.

[10] "Checkstyle — overview," https://checkstyle.sourceforge.io/, Checkstyle, 2025, "Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.".

[11] "Sonarsource rules for java," https://rules.sonarsource.com/java, SonarSource, 2025, includes numerous convention and naming rules (e.g., variable/class/interface naming) categorized as code smells.

[12] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" *arXiv preprint arXiv:1907.00376*, 2019. [Online]. Available: https://arxiv.org/abs/1907.00376

[13] C. E. C. Dantas, A. M. Rocha, and M. A. Maia, "How do developers improve code readability? an empirical study of pull requests," *arXiv preprint arXiv:2309.02594*, 2023, finds SonarQube detected only 26 of 370 real readability improvements, indicating gaps in static tools. [Online]. Available: https://arxiv.org/abs/2309.02594

[14] A. Peruma, E. Murphy-Hill, and C. McMillan, "Ideal: An open-source identifier name appraisal tool," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 634–638.

[15] Y. Zhang, H. Liu, X. Peng, and Z. Xing, "Automatic identifier inconsistency detection using code dictionary," *Journal of Systems and Software*, vol. 191, p. 111398, 2022.

[16] H. Wang, C. Liu, J. Zhao, C. Luo, and X. Zhang, "Deep learning-based identification of inconsistent method names: How far are we?" *Empirical Software Engineering*, 2025, online first.

[17] A. Peruma, C. McMillan, and E. Murphy-Hill, "Towards a model to appraise and suggest identifier names," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–25, 2022.

[18] L. Zhang, M. Wen, Z. Jin, D. Ye, and Y. Xiong, "An accurate identifier renaming prediction and suggestion approach," *Information and Software Technology*, vol. 156, p. 107125, 2023.

[19] S. Vijayvargiya, M. Saad, and T. Sharma, "Enhancing identifier naming through multi-mask fine-tuning of language models of code," in *Proceedings of the 24th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2024, pp. 1–12, replication package available online. [Online]. Available: https://huggingface.co/spaces/scam2024/ReIdentify

[20] F. Liu, X. Xia, D. Lo, Z. Xing, and S. Li, "Learning to recommend method names with global context," *Journal of Systems and Software*, 2022, arXiv preprint. [Online]. Available: https://arxiv.org/pdf/2201.10705

[21] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "Automated variable renaming: Are we there yet?" *Journal of Systems and Software*, vol. 197, p. 111567, 2023.

[22] P. Carbonnelle, "Top ide index," https://pypl.github.io/IDE.html, accessed: 17 November 2025.

[23] W. Anderson, "Extensions using codelens," https://code.visualstudio.com/blogs/2017/02/12/code-lens-roundup, Feb. 2017, visual Studio Code Blog.

[24] "2025 stack overflow developer survey," https://survey.stackoverflow.co/2025, accessed: 17 November 2025.

[25] "Pep 8 – style guide for python code," https://peps.python.org/pep-0008/.

[26] "Java code conventions," https://www.oracle.com/java/technologies/javase/codeconventions-introduction.html.

[27] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring," in *Proceedings of ACL*, 2020, pp. 2699–2712.

[28] Ollama Team, "Ollama," https://ollama.com/, 2023, version information accessed on December 20, 2025.

[29] B. Warner, A. Chaffin, B. Clavié, O. Weller, O. Hallström, S. Taghadouini, A. Gallagher, R. Biswas, F. Ladhak, T. Aarsen, N. Cooper, G. Adams, J. Howard, and I. Poli, "Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference," 2024. [Online]. Available: https://arxiv.org/abs/2412.13663