MACEDON : Supporting Programmers with Real-Time Multi-Dimensional Code Evaluation and Optimization

XUYE LIU, University of Waterloo, Canada
YUZHE YOU, University of Waterloo, Canada
XINRONG QIU, University of Waterloo, Canada
TENGFEI MA, Stony Brook University, USA
JIAN ZHAO, University of Waterloo, Canada

CCS Concepts: • **Human-centered computing → User interface design**; **Natural language interfaces**.

Additional Key Words and Phrases: Code Evaluation and Optimization, Code Generation, Programming Interface, Large Language Models

ACM Reference Format:
Xuye Liu, Yuzhe You, Xinrong Qiu, Tengfei Ma, and Jian Zhao. 2026. MACEDON : Supporting Programmers with Real-Time Multi-Dimensional Code Evaluation and Optimization . 1, 1 (January 2026), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abstract

Recent advancements in Large Language Models (LLMs) have led programmers to increasingly turn to them for code optimization and evaluation. However, programmers need to frequently switch between code evaluation and prompt authoring because there is a lack of understanding of the underlying code. Yet, current LLM-driven code assistants do not provide sufficient transparency to help programmers track their code based on the intended evaluation metrics, a crucial step before aligning these evaluations with their optimization goals. To address this gap, we adopted an iterative, user-centered design process by first conducting a formative study and a large-scale code analysis. Based on the findings, we then developed MACEDON, a system that supports multi-dimensional code evaluation in real time, direct code segment optimization, as well as shareable report displays. We evaluated MACEDON through a controlled lab study with 24 novice programmers and two real-world case studies. The results show that MACEDON significantly improved users' ability to identify code issues, apply effective opti-mizations, and understand their code's evolving state. Our findings suggest that multi-dimensional evaluation, combined with interac-tive, segment-specific guidance, empowers users to perform more structured and confident code optimization. The code for this paper can be found in https://github.com/xuyeliu/MACEDON.

Authors' Contact Information: Xuye Liu, University of Waterloo , Waterloo, Ontario, Canada, x827liu@uwaterloo.ca; Yuzhe You, University of Waterloo , Waterloo, Ontario, Canada, y28you@uwaterloo.ca; Xinrong Qiu, University of Waterloo , Waterloo, Ontario, Canada, a3qiu@uwaterloo.ca; Tengfei Ma, Stony Brook University, Department of Biomedical Informatics , Stony Brook, New York, USA, tengfei.ma@stonybrook.edu; Jian Zhao, University of Waterloo , Waterloo, Ontario, Canada, jianzhao@uwaterloo.ca.

Manuscript submitted to ACM

1 Introduction

The advent of Large Language Models (LLMs) has led to a paradigm shift in AI-driven code assistants [7, 11, 14, 27, 28] and brought transformative changes to programmers' workflow. Due to the lack of expertise, novice programmers heavily rely on LLM-driven code assistants in their workflow to generate or optimize sophisticated code from natural language (NL) prompts [13, 16, 24]. Instead of manually reviewing their code, programmers can now declaratively express their optimization goals to LLMs. However, this approach inadvertently overlooks the need for programmers to understand their code in depth or to conduct goal-driven strategies for opti-mization, which could be challenging for novices when evaluating the quality of their generated code.

Recent research on programmers' interaction with LLM-driven code assistants has reflected these challenges for novice programmers. Specifically, uncertainty in generated code can lead to effi-ciency problems during the development process, as programmers are left to mentally anticipate possible outcomes until the code is generated. While understanding the current state of code is crucial for optimization prompts, there still remains a signifiacnt gap in supporting the complex pro-cess of forming optimization intentions. Therefore, our goal of this paper is explore a design that supports the iterative LLM driven code optimization and evaluation.

To address this gap, we conducted a study on 6 programmers of varying experience who use LLM based code optimization on a regular basis. Out aim was to realize how programmers come up with optimization strategies and the challenges they face. The study revealed the need of a *structured, multidimensional* optimization and evaluation system that will also support selective segment optimization without affecting the whole codebase to a graet extent.

Gaining insight from the study, we focused on designing an LLM based code evaluation and optimization system based on the findings of the following research questions:

RQ1- Evaluation & Strategy. What evaluation dimensions and optimization strategies do programmers adopt when working with LLM-based assistants?

RQ2- Tool Design & Effectiveness. How can we design effective tools that support multi-dimensional code optimization, and to what extent are they useful for real-world programming tasks?

RQ3- Generalizability. Can such tools extend beyond novice programmers and remain effective in more diverse programming scenarios?

To answer these RQs, we began by analyzing a large dataset of over 70,000 real-world code examples from the Performance Improving Edits (PIE) dataset to identify five measurable dimensions for code optimization. This analysis gave insights on **(RQ1)** by helping us understand how programmers evaluate and optimize their code. For anwsering **(RQ2)** we built MACEDON, a VSCode extension that provides multidimensional code evaluation, segment specific suggestions and optimization. In order to justify its effectiveness, we examined it on 24 novice programmers. The tool was built based on three derived design goals: 1) helping users assess their code across the five dimensions using interpretable scores, 2) offering specific suggestions such as improving loop efficiency or renaming unclear variables, and 3) enabling users to apply changes consistently with minimal manual effort. Furthermore, to explore whether MACEDON is useful beyond novice programmers (RQ3), we conducted two case studies in which participants applied the tool to real programming tasks. In summary, our contribution is therefold:

- A formative study and a comprehensive data analysis of optimization behaviors and needs when using LLMs for code evaluation and optimization.
- An interactive tool, MACEDON, developed as a Visual Studio Code Extension, that supports segment-specific code optimizations in multiple dimensions, minimizing rework and improving efficiency.

- A user study and two case studies for assessing MACEDON, demonstrating its effectiveness in improving code quality and user experience compared to conventional LLM-based assistants.

2 Related Work

2.1 Generating and Optimizing Code with LLMs

In recent period, we have seen a huge tendency of both generating and optimizing codes using LLMs like GPT [1], Codex [7], CodeGen [18], and InCoder [10] which can accept requests in NL. Moreover, models like GitHub Copilots can perform code completion and Optimization based on the whole code-base and also provide suggestions [5].

Other LLM-based frameworks have also been developed to address different challenges in code generation and optimization. For instance, ClarifyGPT [17] enhances the code generation process by identifying ambiguities in user prompts and seeking clarifications, ensuring that generated code aligns closely with user intentions. CodePlan [4] addresses the challenge of complex repository-level tasks by using a task-agnostic, neuro-symbolic framework that frames coding as a planning problem, synthesizing a multistep chain of edits through dependency analysis, change impact analysis, and adaptive planning with neural LLMs. SBLLM [12] combines LLMs with search techniques for iterative code optimization, using representative sample selection, adaptive pattern retrieval, and genetic operator-inspired prompting to achieve code efficiency improvements. CoLadder [25] provides a hierarchical structure for decomposing programming tasks, allowing programmers to better align their problem-solving intentions with LLM-generated code. Unlike search-based optimization or task decomposition, our approach emphasizes providing programmers with real-time insights into their code's status through a structured interface that facilitates direct, segment-specific op- timizations. MACEDON uniquely combines visualization of code state with targeted recommendations, enabling programmers to efficiently track and improve code quality through a seamless integration into their workflow.

2.2 Evaluation of LLM-based Code Assistants

Evaluating the quality of code produced by Large Language Models is essential for software development. Traditional benchmarks such as HUMANEVAL [7] and MBPP [3] measure functional correctness by testing generated Python functions against predefined cases. To broaden this scope, MultiPL-E [6] and HumanEval-X [29] translate these benchmarks into various languages for cross-language evaluation. Specialized challenges like AlphaCode [14] utilize Codeforces to test complex problem-solving, while Spider [26] focuses on text-to-SQL tasks. EvalPlus [15] enhances these methods by automatically generating larger sets of test inputs to increase testing coverage. Furthermore, SWE-bench shifts the focus toward real-world software engineering by requiring models to resolve GitHub issues within complex, multi-file codebases.

Unlike these static evaluations, MACEDON emphasizes real-time feedback and iterative refinement during the optimization process. While existing benchmarks primarily assess functional correctness through fixed tests, MACEDON provides a dynamic, user-centered framework. This allows programmers to evaluate code across multiple dimensions and refine it interactively using LLM-driven recommendations. By bridging the gap between one-off generation and continuous improvement, this approach offers a more practical solution for the ongoing nature of real-world programming tasks.

2.3 Traditional Code Analysis and Optimization Tools

Traditional code improvement has evolved through performance optimization and refactoring. Performance analysis tools such as HPCToolkit [2] and Speedoo [8] identify bottlenecks using runtime data, while regression detection systems [19] maintain efficiency. However, these tools often require separate execution phases outside the primary development workflow. Similarly, static analysis and refactoring tools like JDeodorant [9], and IntelliJ [20] automate transformations based on established patterns . Despite their utility in maintaining consistency, they lack the contextual depth and semantic understanding necessary for complex, domain-specific tasks that depend on programmer intent.

Recent machine learning advances have introduced neural program repair and learning-based performance optimization. Research into mixed-initiative IDEs, such as Grounded Copilot [5] and in-IDE generation , explores AI assistance but focuses primarily on generation rather than comprehensive evaluation or optimization workflows. As noted by quantitative assessments of development techniques , a significant challenge remains in providing real-time, multi-dimensional insights into a code's state to help programmers craft effective strategies.

Our work, MACEDON, addresses these gaps by merging real-time code evaluation with interactive optimization. Unlike traditional performance tools that operate in separate phases, MACEDON provides immediate feedback during the coding process. While static tools rely on predefined rules, MACEDON leverages LLMs for context-aware refinements. By integrating evaluation and optimization into a single interface, this approach moves beyond existing tools that treat these essential development steps as isolated processes.

3 Design Process & Goals

We conducted an iterative user-centered design process to create MACEDON. The design process included three key stages: 1) Understanding & Ideation—involving an interview study with experienced programmers to uncover challenges and strategies in code optimization using LLM-driven tools; 2) Prototype & Walkthrough—the design and development of MACEDON, informed by the insights gained, followed by a cognitive walkthrough for feedback and iterative refinements; 3) Deploy & Evaluate—a user study to assess how programmers interact with the system and its perceived usefulness in streamlining code optimization. In this section, we describe the first stage of our design process, outlining the strategies and design goals that guided the development of MACEDON.

3.1 Interview Process

We recruited six participants (4 males, 2 females; ages 20 to 27, $M = 23.5$, $SD = 1.2$) with different levels of programming experiences through purposive sampling for our interviews. A pre-test survey screened for eligibility using a 5-point scale for experience, total years in the field, and self-reported use of AI tools. The group included three novice programmers with approximately one year of experience and two with at least five years. Overall, participants were well-versed in programming ($M = 4.17$, $SD = 0.41$) and frequently utilized LLMs for optimization ($M = 7.5$, $SD = 2.10$ times/week). They provided informed consent and received 20 CAD for a 60-minute session.To encourage reflection on optimization strategies, participants shared recent examples of ChatGPT usage prior to the study. During the 60-minute sessions, researchers conducted interviews exploring the challenges of evaluating code states, translating those evaluations into further optimizations, and identifying specific user needs. All sessions were audio-recorded and transcribed for analysis. The team performed thematic analysis [23] using both inductive and deductive methods. Following two iterations of analysis and the resolution of disagreements through discussion, the researchers identified and categorized the final key themes and participant strategies.

3.2 Interview Results

Here, we present our findings on the workflows that participants adopted during the code optimization process and the challenges they encountered.

3.2.1 Multi-Dimensional Code Evaluation. Participants' code optimization involved two primary phases: assessing the current code state across dimensions like clarity and efficiency to identify shortcomings, and then implementing specific improvements. However, a lack of transparency in recommendations often made it difficult to determine if changes improved performance or merely increased complexity. As P4 noted, distinguishing between actual optimization and added complexity is a significant hurdle. To manage this cognitive load, all participants adopted a strategy of breaking the optimization process into smaller, manageable steps.

All six participants indicated that a structured, multi-dimensional approach would increase efficiency. While some prioritized time efficiency and documentation, others admitted to overlooking space usage and redundancy until prompted by external feedback. P2 and P6 highlighted that issues like space efficiency or code redundancy often go unnoticed unless specifically flagged or until others struggle to read the code. These observations suggest that a systematic, multi-dimensional evaluation framework is necessary to ensure code optimization is both organized and comprehensive.

3.2.2 Facilitating Direct Code Segment Optimization. Participants struggled with managing large volumes of code, often feeling a loss of control during the optimization process. This led to a strong preference for optimizing *"dimension by dimension,"* as noted by P2. The majority of participants (4/6) expressed frustration when systems applied changes to the entire codebase simultaneously. Instead, they preferred the direct manipulation of specific segments based on individual priorities, such as time performance or clarity. P5 highlighted that fixing one section at a time helps in understanding exactly what changes are being made and why.

To address these difficulties, most participants (4/6) utilized a strategy of optimizing self-contained segments independently before merging them back into the larger codebase. This modular approach allowed users like P4 to see improvements gradually. Another common method involved targeting specific code segments based on data from previous runs to avoid unintended effects on other sections. However, this process remains tedious; P5 noted the significant effort required to track, preserve, and compare optimized versions against the original code while ensuring no conflicts arise within the rest of the system.

3.2.3 Real-Time Feedback for Iterative Code Optimization. Every participant expressed frustration with the constant switching between code evaluation and applying optimizations, which frequently caused cognitive overload. P2 shared, *"Sometimes I need to stop and evaluate if the optimization worked; it breaks my flow and makes the whole process slower."* P5 agreed, noting it was *"frustrating to go back and forth between evaluation and deciding whether to make changes."* Despite these challenges, participants favored real-time feedback to quickly verify *"whether the code change is accurate,"* as stated by P4.

Many participants highlighted the advantages of performing iterative optimization on small, individual code segments. P4 explained, *"I often focus on the areas that require the most attention first and do iterative prompting until it works."* This iterative approach to refining specific sections allowed users to maintain control over the development process. By focusing on targeted adjustments rather than broad changes, they ensured that each modification remained strictly aligned with their overall optimization goals without negatively impacting the entire codebase.

3.3 Design Guidelines

Based on the interview results, we derived the following design guidelines to drive the development of MACEDON.

DG1: Providing Multi-Dimensional Feedback for Code Evaluation. A lack of comprehensive evaluation across different metrics can prevent programmers from fully understanding the state of their code. The system should offer multi-dimensional feedback,such as performance, readability, and clarity, to help users better understand the strengths and weaknesses of their code. This evaluation should be presented in a structured and organized manner, allowing programmers to externalize their thought processes and refine specific areas of the code based on the feedback. Flexible feedback options should reflect the various dimensions programmers need for code optimization.

DG2: Facilitating Direct Code Segment Optimization. Programmers often feel a loss of control when working with large amounts of code that require detailed optimization. The system should support direct manipulation of specific code segments, enabling users to select and modify areas for improvement based on their priorities. It should also allow programmers to reorganize or refine their code incrementally, providing the ability to test and apply optimization recommendations to one segment at a time, instead of overwhelming them with global code changes.

DG3: Integrating Real-Time and Interative Code Evaluation with Optimization Suggestions. Programmers often face cognitive overload due to the constant need to switch between code evaluation and applying optimizations. The system should integrate real-time feedback with iterative suggestions, allowing programmers to continuously assess their code and apply optimizations without disrupting their workflow. By providing timely, contextual feedback during the optimization process, the system can guide users in refining their code in a smooth, iterative manner.

4 Define Code Evaluation Method

The insights of our formative study led to the investigation of our **RQ1** regarding evaluation dimensions and optimization strategies that programmers normally use. As indicated by DG1, we decided to evaluate the C++ code on five key metrics: redundancy, documentation, clarity, time efficiency, and space efficiency.

4.1 Data Collection

The researchers utilized the Performance-Improving Edits (PIE) dataset [22], which consists of human-programmer optimizations from CodeNet [21] competitive programming tasks. C++ was chosen as the target language because of its importance in performance-critical applications and its compatibility with the Gem5 simulator. Since C++ submissions often focus on fine-grained optimizations, they provide an ideal basis for analyzing time and space efficiency metrics.

To maintain high data quality, the team filtered submissions to include only functionally correct programs that passed all test cases and runtime constraints. Each program was then paired with its improved version for comparative analysis. By using the Gem5 simulator, the researchers normalized execution times to remove inconsistencies in raw runtime data. The final dataset comprises 77,967 program pairs across various problem domains.

4.2 Data Analysis and Results

4.2.1 High-Quality Code Characteristics. We began the analysis by describing the general characteristics of these C++ codes and deciding what parameters needed to be used in a five-dimensional framework. Programs in the PIE dataset

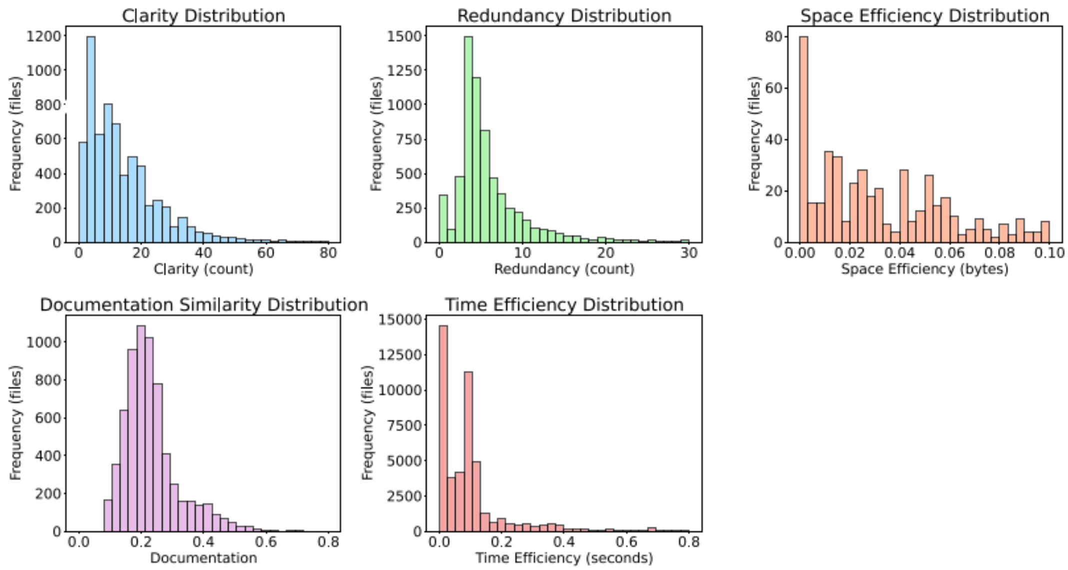

Fig. 1. Distributions of key metrics: clarity, redundancy, time efficiency, space efficiency, and documentation in PIE dataset.

demonstrate strong performance across all five optimization dimensions (Figure 1). Most programs exhibit high clarity (fewer than 10 issues per file) and low redundancy (fewer than 5 redundant constructs), indicating well-structured and efficient code. Space and time efficiency metrics cluster around optimal values, while documentation scores fall within moderate ranges (0.1-0.3).

4.2.2 Multi-Dimensional Code Quality Analysis. Table 1 presents our analysis of five key metrics across the PIE dataset. The results reveal distinct patterns: **Clarity:** 58.65% of files contain fewer than 10 clarity-related issues, indicating generally well-structured code. **Redundancy:** 32.36% of files show moderate to high levels of redundant constructs, suggesting optimization opportunities. **Space Efficiency:** 19.19% of files consume significantly more memory than average, highlighting areas for memory optimization. **Documentation:** 6.30% of files demonstrate high semantic alignment between code and documentation, while 5.60% show minimal documentation efforts. **Time Efficiency:** 1.41% of files achieve exceptional performance, representing solutions that excel across all metrics.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[3] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[4] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 675–698.

[5] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

Table 1. Evaluation methods, formulas, and optimization examples for each dimension. Notation: $perc(.)$ returns a training-set percentile rank in $[0, 1]$; T_{exec} is execution time; M_{usage} is peak memory usage; $Reduntant_{lines} countsduplicate/deadlines$. For clarity, f_i are style factors with weights $w_i (\Sigma_i w_i = 1)$. For documentation, S_{sim} is the cosine similarity between a code segment and its accompanying natural language documentation (e.g., CodeBERT embeddings). The ceiling operator $\lceil . \rceil$ maps the 0-1 value onto the common 1-10 rubric (top 10% receive 10).

Metric	**Formula**	**Evaluation Method**	**Examples(Original → Optimized)**
Time Efficiency	$Score = \lceil 10(1 - perc(T_{exec})) \rceil$	Algorithmic complexity and runtime behavior	$pow(x, 2) \longrightarrow x * x$
Space Efficiency	$Score = \lceil 10(1 - perc(M_{usage})) \rceil$	Memory footprint and data structure usage	vector<vector<int» mat(n, vector<int>(n,0))⟶ vector<int>(n*n,0)
Clarity	$Score = \lceil 10(1 - \Sigma_i w_i perc(f_i)) \rceil$	Magic literals, naming consistency, and statement length Descriptive names and magic number avoidance	int a=0;⟶ int sum=0; int f(int x);⟶ int computeSquare(int x)
Documentation	$Score = \lceil 10(perc(S_{sim})) \rceil$	Embedding-based semantic similarity between code and documentation	// calculate result ⟶ // Computes total revenue from all entries
Redundancy	$Score = \lceil 10(1 - perc(Redundant_{lines})) \rceil$	Duplicate code and dead code detection	for(...) print(x); for(...) print(x);⟶ for(...) print(x);

[6] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3675–3691.

[7] Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Zhifei Chen, Bihuan Chen, Lu Xiao, Xiao Wang, Lin Chen, Yang Liu, and Baowen Xu. 2018. Speedoo: prioritizing performance optimization opportunities. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE 2018).

[9] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. 2007. Jdeodorant: Identification and removal of feature envy bad smells. In *2007 ieee international conference on software maintenance*. IEEE, 519–520.

[10] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[11] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. *URL https://github. blog/2021-06-29-introducing-github-copilot-ai-pair-programmer* (2021).

[12] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. 2024. Search-based llms for code optimization. *arXiv preprint arXiv:2408.12159* (2024).

[13] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–19.

[14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[15] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2023), 21558–21572.

[16] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

[17] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[18] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[19] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–30.

[20] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. 2024. Together we go further: Llms and ide static analysis for extract method refactoring. *arXiv preprint arXiv:2401.15298* (2024).

[21] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021).

[22] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867* (2023).

[23] Gareth Terry, Nikki Hayfield, Victoria Clarke, Virginia Braun, et al. 2017. Thematic analysis. *The SAGE handbook of qualitative research in psychology* 2, 17-37 (2017), 25.

[24] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[25] Ryan Yen, Jiawen Stefanie Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2024. Coladder: Manipulating code generation via multi-level blocks. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.

[26] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

[27] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7443–7464.

[28] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).

[29] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5673–5684.