Grounding Generative AI in Software Engineering: Are We There Yet?

Mootez Saad
Dalhousie University
Halifax, Canada
mootez@dal.ca

José Antonio Hernández López
University of Murcia
Murcia, Spain
joseantonio.hernandez6@um.es

Boqi Chen
McGill University
Montréal, Canada
boqi.chen@mail.mcgill.ca

Neil Ernst
University of Victoria
Victoria, Canada
nernst@uvic.ca

Dániel Varró
Linköping University
Linköping, Sweden
daniel.varro@liu.se

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

***Abstract*—Large Language Models (LLMs) have made significant contributions to software engineering, particularly in the field of code generation, demonstrating the ability to produce functionally correct code snippets. The development of these models involves multiple stages, including pre-training on large amounts of source code data and aligning them with human preferences using various techniques. However, their development process often neglects foundational software engineering (SE) practices and principles. Specifically, LLMs receive limited exposure to core SE concepts during training, such as modularity, single responsibility, cohesion, and coupling. As a result, the generated code may lack the properties critical for building maintainable, extensible, and robust software systems.**

This vision paper advocates integrating SE knowledge directly into LLMs to enhance their capability to generate code and SE artifacts that adhere to and align with established best practices. We propose a new direction for LLMs to move beyond their current focus on functional accuracy toward producing robust, maintainable software. To assess how well LLMs internalize SE knowledge, we propose adopting Bloom's Taxonomy as a comprehensive assessment framework, offering a structured alternative to limited evaluation methods such as probing. By embedding software engineering principles, next-generation LLMs can leverage decades of software engineering knowledge and transform software development with reliable, high-quality generative capabilities.

I. Introduction

Large language models (LLMs) are transforming software development by supporting various software engineering (SE) tasks such as on-demand code and test generation, code comprehension and summarization, code completion, and assistance capabilities [1], [2]. By learning from vast code repositories, LLMs have become valuable tools for developers, potentially enhancing productivity and reducing the time required to write boilerplate or routine code.

An important aspect of source code is that it is rarely written in isolation. Rather, it must conform to constraints, standards, and best practices. These constraints stem from the necessity for code to integrate seamlessly with existing components of the software system, be understandable and maintainable by other developers, and remain adaptable to future requirements. Following best practices, such as design principles, leads to high-quality code that exhibits high maintainability and extensibility, for instance. Therefore, evaluation of developed or generated source code goes beyond mere functional correctness [3].

However, while LLMs are used within a software engineering context, they are not entirely trained on fundamental software engineering knowledge embodied in the field in the form of, for example, best practices and design principles. Most contemporary models are *directly* pre-trained on *static* source code, and mainly target functional correctness. However, source code is the end product of an engineering process that includes deliberation and design which are not directly captured in the source code itself. Currently, code LLMs are trained to autocomplete functionally correct code. This creates a rift between SE expectations and what these models deliver. This deficit in expectation was initially investigated by Pudari and Ernst [4]. They created a hierarchy of software abstractions inspired by Koopman's Autonomous Vehicle Safety Hierarchy of Needs [5]. Using this hierarchy, they showed that Copilot [6] could clear out low levels of abstractions, *e.g.,* autocompleting syntactically correct code, but often struggled with deducing and suggesting the correct design patterns and architectural tactics for a given project. Similarly, Cao *et al.* [7] demonstrated recent models such as WizardCoder [8] and DeepSeek-coder 33B [9] struggled in programming scenarios involving object-oriented programming concepts, notably *inheritance* and *encapsulation*, and were even outperformed by undergraduate students. Recently, Pan *et al.* [10] have shown that models such as GPT-4o and Llama 3.1-70B tend to struggle in classifying design patterns, achieving an overall accuracy of 38.81% when tested to recognize different structural and creational design patterns.

This challenge extends beyond code generation to more abstract software engineering tasks such as domain modeling. In their work, Chen *et al.* [11] showed that although GPT-4 achieves a relative high performance in generating classes given a textual domain description (F_1 = 0.76) and their attributes ($F_1 = 0.61$), it struggles to model the correct relationships ($F_1 = 0.34$). Furthermore, it rarely applies modeling best practices, notably the Abstraction-Occurrence pattern [12].

Given this evidence, it is reasonable to worry that they may produce poorly structured software designs, leading to inflexible, hard-to-maintain systems. This could increase technical debt and maintenance costs. This, in turn, presents a significant drawback, given that code quality is one of the main drivers for developer productivity [13].

In this paper, we start by providing a brief overview of the current practices involved in pre-training and evaluating large language models of code. We then lay out a vision for how these models can be further aligned with SE practices. We propose the integration of higher-level design and architectural insights into the training process to equip language models with a deeper understanding of software systems and with better reasoning capabilities. In addition, we propose the incorporation of other evaluation strategies to validate these models that extend beyond functional correctness. Furthermore, understanding how and to what extent LLMs acquire SE knowledge is vital for trusting their outputs. We discuss current model interpretation methods, their limitations, and potential alternatives. Specifically, using code generation as a starting point, we demonstrate the use of Bloom's taxonomy to structurally probe the SE knowledge that the current set of LLMs internalize.

II. The Status Quo

This section provides background knowledge by presenting current practices used to pre-train and evaluate, and analyze LLMs for code.

A. Pre-training of LLMs of Code

1) Pre-training objectives: LLMs are typically trained on extensive text-based datasets such as *The Pile* [14] and *FineWeb* [15] that include source code and question-answer discussions involving code snippets. Language models of code are trained specifically using source code without explicitly taking code quality into consideration [16]. Pre-training objectives enable them to learn patterns, structures, and semantics to predict code-related properties and to generate code effectively.

Common pre-training objectives, used by the GPT [17]-family of models, include next-token prediction where they predict the next word based on previous ones, modeling language in a left-to-right manner.

Models like CodeT5 [18], specialized for source code, extend these objectives. It consolidates pre-training tasks tailored for code understanding and generation into a single framework. Specifically, it combines masking strategies by masking code spans and identifiers and trains the model to predict them, enhancing its understanding of code syntax and semantics. It also includes dual-generation tasks between code and natural language, training on code-to-text (*e.g.,* generating documentation from code) and text-to-code (*e.g.,* generating code from natural language descriptions), to bridge the gap between code and human language.

Recently, models such as CodeLlama [19] and StarCoder 2 [20], are pre-trained using the fill-in-the-middle (FIM) objective. In this objective, the model is trained to generate a missing code segment between a given prefix and suffix. With this, the model becomes adept at understanding bidirectional context and generates code that integrates with the surrounding snippet. This enhances its capability in tasks like code completion, refactoring, and debugging.

2) Data representation: Another important aspect during pre-training is the input representation fed to the language model. CuBERT [21], one of the earliest code language models, was trained directly on Python code snippets, treating it as a plain sequence of tokens. CodeBERT [22] improved upon this by introducing a bimodal representation where the input consists of a natural language sequence that describes the code snippet, followed by the code snippet. This approach aims to capture the semantic relationship between natural language and programming languages, particularly relevant for tasks such as code search and documentation generation.

However, treating code as a stream of tokens ignores its inherent syntactic and semantic structure, failing to capture the relations and dependencies within it properly. To address this limitation, GraphCodeBERT [23] enriches the input representation by incorporating data flow graphs of the code alongside the token sequences. This enhanced representation allows the model to comprehend the semantic nuances of the code more effectively, which enhances its performance on tasks that require advanced code understanding.

While the aforementioned models show good results in various code intelligence tasks, they still operate primarily at the function or file level. However, real-world scenarios involve complex dependencies with interactions between different parts of the codebase. By confining the pre-training phase to isolated code snippets, models may perform suboptimally when code generation necessitates repository-level knowledge. Recognizing this limitation, recent approaches, such as StarCoder 2 [20] and DeepSeek-Coder [24], have begun to include repository-level contextual information during the pre-training stage.

B. Benchmarks and Evaluation

In this section, we cover some of the metrics used to quantify the performance of LLMs of code. Second, we list a non-comprehensive list of benchmarks used to systematically evaluate them.

1) Performance metrics: The evaluation metrics used can be broadly categorized into two types: *token-based* metrics and *execution-based* metrics. Token-based metrics, such as BLEU [25], ROUGE [26] and CodeBLEU [27], evaluate the generated code by comparing it to reference code at the token or syntactic level. They quantitatively assess similarity in terms of syntax and structure. However, these metrics may not fully capture functional correctness or the ability of the code to execute successfully. Execution-based metrics, such as *pass@k* and *execution accuracy*, assess the functional correctness by executing the generated code and verifying the outputs against expected results, usually against test suites.

2) Benchmarks: Several widely used benchmarks have been developed to evaluate language models of code, *inter alia*, HumanEval [6], MBPP [28], and recently, BigCodeBench [29] and SWE-bench [30]. HumanEval consists of 164 handcrafted Python programming problems, each providing a natural language description, a function signature, and hidden unit tests. Models are evaluated based on their ability to generate code that fulfills the problem descriptions and passes all the hidden tests, using metrics like pass@k to measure functional correctness. MBPP contains 974 crowd-sourced programming tasks designed to reflect challenges encountered by beginner programmers. It assesses models on both syntactic correctness and functional performance across a diverse set of basic programming tasks. BigCodeBench developed as part of the BigCode project, is designed for HumanEval-like function-level code generation tasks but with more complex instructions and diverse function calls to simulate solving practical problems faced in real-world scenarios.

SWE-bench [30] has been proposed as a more realistic benchmark that includes tasks beyond generating code from natural language queries. Specifically, it includes GitHub issues collected from open source Python projects, and evaluates LLMs capacity in solving them.

While more realistic, benchmarks like SWE-bench still define success as passing the provided tests, not the quality, maintainability, or in general, the long term effects of the generated patch.

C. Model Understanding

Probing [31] is a technique used to evaluate whether LLMs have implicitly learned a specific property during (pre-) training. The core idea is to train lightweight classifiers on top of the model's hidden representations to predict the property of interest. High classification accuracy suggests that the LLM internally encodes that property. For example, Tegmark and Gurnee [32] demonstrate that Llama 2 models embed concepts of time and space within their internal representations. In the domain of code, probing has been employed to determine whether LLMs capture specific code-related properties [33], [34], [35]. For instance, López et al. [34] successfully extracted complete abstract syntax trees from models such as CodeBERT, GraphCodeBERT, and CodeT5 using classifiers. They found that a low-dimensional syntactic subspace exists within the hidden representation of these models that encodes the complete abstract syntax trees of input code snippets across all considered models. Such a subspace is found within the representations of middle layers where the probe's performance peaks, whereas it declines in earlier and later layers. These findings indicate that models encode syntactic information. However, to the best of our knowledge, probing has not been used to determine other dimensions of code such as quality.

In summary, the current status quo in pre-training, evaluation and understanding of code LLMs is heavily skewed towards syntactic validity and functional correctness. The community has successfully built models that can generate code and pass tests. However, this focus has left a critical gap: the *engineering* quality, the principles that guarantee maintainability and other attributes are not explicitly incorporated. This confirms the *rift* between the needs of software engineering and current models and necessitates a new approach which we present in our vision.

III. Vision: Software Engineering Grounded Language Models

So far, LLMs are conceived as "(natural) language" models, and hence natural language is the first-class citizen. Given the textual form, these models perform code-related tasks satisfactorily. Most of LLM development, during architecture conceptualization, pre-training, or evaluation, focuses on generating syntactically and functionally correct code. However, as discussed in Section I, focusing on functional correctness is necessary, but not sufficient.

SE *native* models are grounded in SE and can adhere to SE principles and software systems. We need SE native large language models because the existing AI models overlook the essential SE knowledge encapsulated in the form of principles and best practices to make software maintainable, extensible, robust, and reliable. In the following subsections, we lay out a vision, illustrated in Figure 1, of how to make these models *grounded in* SE. We discuss some of the works conducted towards this goal, their limitations, and potential new directions to address these limitations.

A. Enriching LLMs with SE Knowledge

To transform language models into truly SE-grounded tools, it is important to extend their training beyond the generation of syntactically correct and functionally accurate code. While these attributes are necessary, they do not encompass the full spectrum of SE practices that ensure code is maintainable, extensible, and aligned with sound design principles. We propose several strategies as a step towards achieving this integration.

1) SE Guided Training: Software development processes produce many artifacts. Including these artifacts in training can potentially enhance the models' capabilities. For example, Unified Modeling Language (UML) diagrams provide abstract high-level representations of software systems, capturing essential aspects such as class hierarchies, object interactions, and system behaviors. Training models on both the code and its corresponding abstractions would enable them to understand the relationships between high-level design and concrete implementations. This multimodal learning strategy would allow models to grasp architectural patterns and design principles that are not readily apparent from code alone. It can bridge the gap between conceptual understanding and practical application.

While static code provides a snapshot, the engineering process itself is captured in the *evolution* of a codebase. The history of a repository, represented by its sequence of commits,

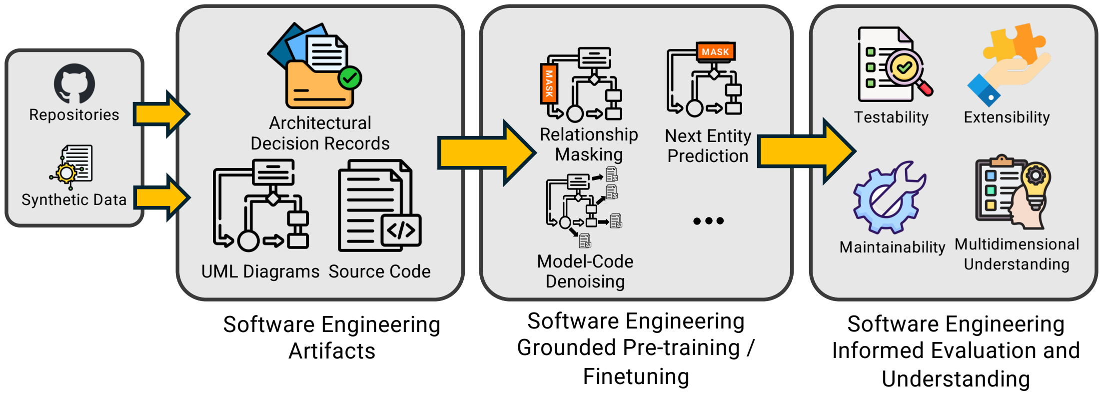

Fig. 1. Overview of the vision of SENAI, incorporating software engineering knowledge into the training and evaluation process of large language models.

provides a readily available dataset of software engineering in action. We propose continuing the training of these models on elements that capture the dynamics of this evolution, such as (code before, diff, commit message) triplets. This would teach the models the semantics of common engineering tasks, *e.g.,* what a refactoring to improve cohesion looks like, or how a bug fix is practically applied. This type of grounding provides a data-rich, practical complement of the static, high-level abstractions discussed above. In addition, it has the potential to equip the model with an internal *software world model* that allows it to explore aspects other than code and simulate an outcome before it can produce an action. This extends similar recent efforts, such as Code World Model [36] that go beyond training on surface-level source code.

Another avenue to reinforce SE knowledge in the models is modifying pre-training objectives to include the prediction of entity relationships within code. By training models to recognize and predict relationships such as composition and association among classes and objects, we enhance their understanding of design and architectural structures. Similar to how models such as CodeT5 predict identifiers and their types, and GraphCodeBERT predicts data flow edges, incorporating entity relation prediction tasks helps models internalize the structural and semantic aspects of code. This enhancement can lead to code generation that not only functions correctly but also adheres to sound architectural practices such as promoting better cohesion and reducing coupling within the codebase. Reinforcement learning (RL)-based techniques can further align code generation models. Incorporating RL guides the models to learn from feedback grounded in SE knowledge. A work towards this direction by Weyssow *et al.* [37] integrates RL into code generation models to better align them to code preferences, such as code readability and style.

2) Beyond Code and Its Abstractions: Incorporating other types of software artifacts, notably, architectural decision records (ADRs), can equip language models with *reasoning abilities* regarding design choices. ADRs document the rationale behind architectural decisions, capturing the context, alternatives considered, and implications of each choice. When adding ADRs to the training data, models could learn to associate code implementations with the underlying design intentions. This insight enables models to generate code that aligns with specific architectural tactics and to provide explanations for their design decisions, mirroring the reasoning process of experienced software engineers.

3) Potential Challenges: We acknowledge that these approaches present challenges, particularly concerning data procurement. Artifacts such as UML diagrams and ADRs are not as widely available as raw source code is, which could limit the volume and diversity of training data. However, there are viable strategies to mitigate this issue. Existing works in architectural recovery provide methodologies for extracting architectural information from codebases, effectively generating the necessary abstractions. Additionally, synthetic data generation techniques, such as those employed in models like WizardCoder [8], can augment the training dataset.

We are not advocating for the abandonment of current data and pre-training methodologies. Instead, we propose enriching existing models with SE-grounded data and training objectives where the signal of software engineering knowledge is more *amplified*. This integration strengthens design and architectural capabilities without compromising code generation strengths, ensuring models produce functional, syntactically correct code that also adheres to software design best practices to engineer reliable software.

Looking ahead, such enrichment becomes particularly crucial for AI Agents [38] orchestrating multi-step software engineering workflows. Agents inherit their underlying LLMs' capabilities and limitations. When models lack architectural awareness (*e.g.,* of coupling or design patterns), agents propagate poor structural choices across generated artifacts. Conversely, SE-grounded LLMs equip agents with built-in architectural validation: preserving abstraction boundaries during refactoring, applying design principles in code generation, and avoiding anti-patterns. This amplification of SE signals in LLMs, can enable agents to evolve from functional code producers to trustworthy engineering partners.

B. Evaluation of LLMs for Code

The evaluation process needs to be adapted to gauge the extent to which the models, old and new, grasp SE knowledge. SE quality attributes such as modularity, cohesion, coupling, and abstraction are crucial for creating maintainable, extensible, and efficient software systems [39]. Importantly, this would make the evaluation more aligned with industry interests. Wan *et al.* [40] surveyed 32 practitioners from 21 organizations across three continents on the topic of software architecture. Coupling and cohesion were frequently discussed as important principles in their interviews. In addition, most participants emphasized the importance of extensibility and maintainability as the non-runtime quality attributes of software systems.

Assessing language models' performance concerning such quality attributes requires a shift in both our evaluation methods and the datasets we use for assessment. Current evaluation benchmarks and metrics primarily measure the correctness of generated code against a set of test cases or reference implementations. They do not assess whether the code adheres to good software design practices. As a result, models might receive high scores even if the generated code is poorly structured or violates fundamental engineering concepts.

These new benchmarks would present tasks that require the model to demonstrate good design practices. For instance, tasks could involve, *refactoring exercises*, where models are provided with low-quality code snippets (*e.g.,* from a maintenance perspective), and tasked to refactor them to improve quality metrics such as *lack of cohesion of methods* (LCOM) and *coupling between objects* (CBO), while maintaining functional correctness. A step further would require the model to first identify the suboptimal parts of code in a software system, and then suggest refactoring strategies that cover different levels of granularity (*i.e.,* from methods to packages).

C. Model understanding

Investigating the SE knowledge internalized by LLMs is an important step towards native SE models. Probing methods are employed to investigate the information that language models have internalized during pre-training. Although useful for identifying linearly separable patterns in model representations, they have a few limitations: a) they risk conflating the probe's simplicity with the model's actual knowledge [41], and provide no evidence of causal use of that knowledge in downstream tasks, b) their results may reflect dataset artifacts rather than genuine model understanding, complicating interpretability, and c) finally, these methods cannot be applied on closed-source models.

The emergence of instruction-based language models, which allow for more flexible and interactive engagement, presents an opportunity to complement existing probing techniques with mature frameworks from other fields. A possible framework would be *Bloom's Taxonomy* [42] which categorizes cognitive skills into hierarchical levels. This framework has been previously used by Buckley and Exton [43] to assess developers' comprehension of a software system. In the context of LLM of code, it has recently been employed by Guerra and Ernst [44] to investigate its capabilities in understanding, reproducing, and creating structures within the VIPER architecture that is used to develop iOS applications. A comprehensive assessment strategy can be designed to probe the model's knowledge across various cognitive dimensions. This framework would allow us to move beyond the surface-level analysis provided by classifier probes. This includes assessing higher-order cognitive skills such as applying principles in new situations, analyzing code for adherence to best practices, evaluating design decisions, and even creating original code that embodies SE principles.

In addition, a taxonomy-based assessment can highlight specific areas where the model excels or underperforms. For instance, a model might effectively recall definitions but struggle with applying principles in code refactoring tasks. Identifying such gaps is crucial for guiding further pre-training and improvement. In Table I, we present an example of a survey that can be used to examine to what extent a model internalizes foundational concepts, notably, cohesion and coupling.

TABLE I
BLOOM'S TAXONOMY LEVELS AND CORRESPONDING POTENTIAL ASSESSMENT QUESTIONS

Level	**Assessment strategy**
Recall	This level relates to recall from learned material. We have decided to ignore this level as it would not gauge useful information. Prompting the model on whether it recollects cohesion or coupling has limited practical implications.
Understand	To assess LLMs' capabilities concerning this level, the model can be prompted with code snippets, or pairs of code snippets, that exhibit different degrees (low and high) of cohesion.
Apply	In this stage, Code snippets with low cohesion and high coupling can be provided to the models for refactoring to improve these attributes. The software's behavior should remain the same. The evaluation will be based on test suite execution and relevant software metrics (*e.g.,* LCOM, CBO).
Analyze	At the analysis level, the model is given a tuple of code fragments with varying degrees (low to high) of cohesion (or coupling) in random order. The model then ranks the snippets from lowest to highest cohesion (or coupling). Also, this level can investigate the model's understanding concerning different types of cohesion (*e.g.,* functional, sequential, communicational) and coupling (*e.g.,* temporal, data).
Evaluate	At this taxonomy level, the model will be given a set or pairs of classes, ranked by cohesion or a coupling measure, and asked to evaluate whether such ranking is valid.
Synthesis	In this stage, the model will be presented with requirements of a feature that must be implemented as a part of a software system. Then, the generated implementation will be evaluated based on correctness and its ability to produce loosely coupled and highly cohesive code.

IV. CONCLUSIONS

Grounding LLMs in SE principles is essential for advancing these models beyond mere functional correctness. In this work, we called for integrating high-level design insights into language models which would steer towards the generation of code that is maintainable, extensible, and aligned with best practices. This proposal bridges the gap between code autocompletion and the demands of SE, leading to AI systems that can more effectively assist developers. Ultimately, this integration holds the promise of enhancing software quality and reducing potential technical debt in real-world applications. In parallel, we have also illustrated potential improvements to the current benchmarks, evaluation practices and analysis methods of language models of code.

REFERENCES

[1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, " Large Language Models for Software Engineering: Survey and Open Problems ," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023.

[2] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, 2024.

[3] "Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," Standard ISO/IEC 25010:2011, International Organization for Standardization, Geneva, Switzerland, mar 2011.

[4] R. Pudari and N. A. Ernst, "From copilot to pilot: Towards ai supported software development," *arXiv preprint arXiv:2303.04142*, 2023.

[5] P. Koopman, "Maturity levels for autonomous vehicle safety," apr 2022. Safe Autonomy Blog.

[6] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[7] J. Cao, Z. Chen, J. Wu, S.-C. Cheung, and C. Xu, "Javabench: A benchmark of object-oriented code generation for evaluating large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024.

[8] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," in *The Twelfth International Conference on Learning Representations*, 2024.

[9] D. AI, "Deepseek coder 33b instruct." https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct, 2023. Accessed: 2025-Jan-16.

[10] Z. Pan, X. Song, Y. Wang, R. Cao, B. Li, Y. Li, and H. Liu, "Do code llms understand design patterns?," *arXiv preprint arXiv:2501.04835*, 2025.

[11] K. Chen, Y. Yang, B. Chen, J. A. Hernández López, G. Mussbacher, and D. Varró, "Automated domain modeling with large language models: A comparative study," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2023.

[12] T. Lethbridge and R. Laganiere, *Object-oriented software engineering: Practical software development using UML and java*. Maidenhead, England: McGraw Hill Higher Education, 2 ed., Dec. 2004.

[13] L. Cheng, E. Murphy-Hill, M. Canning, C. Jaspan, C. Green, A. Knight, N. Zhang, and E. Kammer, "What improves developer productivity at google? code quality," ESEC/FSE '22, ACM, 2022.

[14] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[15] G. Penedo, H. Kydlícek, L. B. Allal, A. Lozhkov, M. Mitchell, C. A. Raffel, L. von Werra, and T. Wolf, "The fineweb datasets: Decanting the web for the finest text data at scale," in *Annual Conference on Neural Information Processing Systems*, 2024.

[16] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.

[17] A. Radford, "Improving language understanding by generative pre-training," 2018.

[18] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "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*, 2021.

[19] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[20] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[21] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International conference on machine learning*, pp. 5110–5121, PMLR, 2020.

[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020.

[23] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *9th International Conference on Learning Representations*, 2021.

[24] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[25] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, 2002.

[26] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, (Barcelona, Spain), Association for Computational Linguistics, July 2004.

[27] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[28] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[29] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, *et al.*, "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," in *The Thirteenth International Conference on Learning Representations*, 2025.

[30] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?," in *The Twelfth International Conference on Learning Representations*, 2024.

[31] A. Ettinger, A. Elgohary, and P. Resnik, "Probing for semantic evidence of composition by means of simple classification tasks," Association for Computational Linguistics, Aug. 2016.

[32] W. Gurnee and M. Tegmark, "Language models represent space and time," in *The Twelfth International Conference on Learning Representations*, 2024.

[33] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, 2022.

[34] J. A. Hernández López, M. Weyssow, J. S. Cuadrado, and H. Sahraoui, "Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models," ASE '22, 2023.

[35] A. Karmakar and R. Robbes, "Inspect: Intrinsic and systematic probing evaluation for code transformers," *IEEE Transactions on Software Engineering*, 2024.

[36] Q. Carbonneaux, G. Cohen, J. Gehring, J. Kahn, J. Kossen, F. Kreuk, E. McMilin, M. Meyer, Y. Wei, D. Zhang, *et al.*, "Cwm: An open-weights llm for research on code generation with world models," *arXiv preprint arXiv:2510.02387*, 2025.

[37] M. Weyssow, A. Kamanda, and H. Sahraoui, "Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences," *arXiv preprint arXiv:2403.09032*, 2024.

[38] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, Mar. 2024.

[39] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, 3rd Edition*. Sep 2012. Accessed: 2025-Jan-16.

[40] Z. Wan, Y. Zhang, X. Xia, Y. Jiang, and D. Lo, "Software architecture in practice: Challenges and opportunities," ESEC/FSE '23, ACM, 2023.

[41] Y. Belinkov, "Probing classifiers: Promises, shortcomings, and advances," *Computational Linguistics*, 04 2022.

[42] B. S. Bloom, M. D. Engelhart, E. J. Furst, W. H. Hill, and D. R. Krathwohl, *Taxonomy of educational objectives*, vol. 2. Longmans, Green New York, 1964.

[43] J. Buckley and C. Exton, "Bloom's taxonomy: a framework for assessing programmers' knowledge of software systems," in *11th IEEE International Workshop on Program Comprehension, 2003.*, pp. 165–174, 2003.

[44] L. Guerra and N. Ernst, "Assessing llms for front-end software architecture knowledge," *arXiv preprint arXiv:2502.19518*, 2025.