CHALMERS
UNIVERSITY OF TECHNOLOGY

Literate Programming with LLMs? - A Study on Rosetta Code and CodeNet

Citation for the original published paper (version of record):
Sun, S., Staron, M. (2025). Literate Programming with LLMs? - A Study on Rosetta Code and CodeNet. IEEE Transactions on Software Engineering, In Press.
http://dx.doi.org/10.1109/TSE.2025.3629828

(article starts on next page)

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3629828

Literate Programming with LLMs? - A Study on Rosetta Code and CodeNet

Simin Sun and Miroslaw Staron

***Abstract*—Literate programming, a concept introduced by Knuth in 1984, emphasized the importance of combining human-readable documentation with machine-readable code as writing literate programs is a prerequisite for software quality. Our objective with this paper is to evaluate whether generative AI models, Large Language Models (LLM) like GPT-4, LLaMA or Falcon, are capable of literate programming because of their extensive use in software engineering. To truly achieve literate programming, LLMs must generate natural language descriptions and corresponding code with aligned semantics based on user prompts. In addition, their internal representation of programs should allow us to recognize both programming languages and their descriptions. To evaluate their capabilities, we conducted a study using the Rosetta Code and CodeNet repositories. We perform four computational experiments using the Rosetta Code repository, encompassing 1,228 tasks across 926 programming languages, and validate our findings on the larger CodeNet dataset, which includes 55 tasks and 52 languages. Our findings show that LLMs in the trillion-parameter class are capable of literate programming, while models in the million- and billion-parameter classes are better at recognizing programming languages than tasks. Based on these results, we conclude that modern LLMs inhibit a deeper ability to encode programming languages and the semantics of programming tasks, bringing us closer to realizing the full potential of literate programming.**

***Index Terms*—Large Language Model(LLM), Literate Programming, Computation Experiment, Code-related Tasks**

I. Introduction

LITERATE programming started as an idea to write programs that are understandable both by humans and computers [1]. Instead of writing programs that include comments, programmers should write programs together with their descriptions, for example, using a higher-level language called WEB by Knuth. Such a description would then be input into two processes: weaving, which generates a text, natural language, description of the program, and tangle, which generates a program source code. These two outputs are then compiled into a text document (e.g., using the TeX compiler) and an executable (e.g., using the Pascal compiler). The process is illustrated on the left-hand side of Fig. 1.

Although this idea has been around since 1984, it has been actualized with the introduction of Large Language Models (LLMs), as this process resembles using LLMs for generative programming. The programmers provide a prompt that the LLM completes with a program and (depending on the context) a description of the program/comments in the program's source code. The programmers do not write the program entangled with the description, as in the WEB system, but expect the LLM to describe the program that the LLM generates from the prompt. The programmers rely on the LLMs to generate and explain the code, which means they rely on the model's ability to explain the programs sufficiently. Therefore, the gray background indicates that the two forms must be equivalent. Therefore, we can formulate the literate programming hypothesis for LLMs in the following way: *an LLM is literate in programming when we can use it to explain/recognize the programming language used in the program and the task that this program implements equally well.*

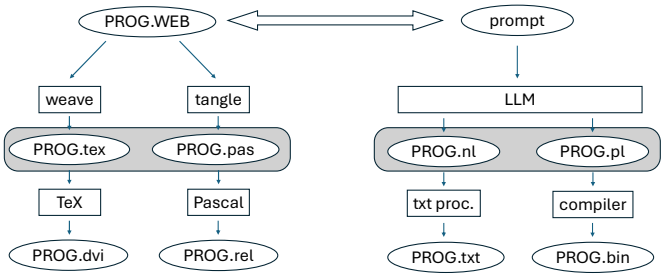


Fig. 1: The WEB system that implements the idea of literate programming by Knuth (left-hand side) and its modification in the context of LLMs (right-hand side). The grey background indicates that two different representations must be semantically equivalent. The WEB systems description of the programs is equivalent to the LLMs prompt, which starts "producing" the outputs.

This hypothesis can be evaluated in two ways. The first is to analyze embeddings: given a program, we derive its programming language (PL) and programming task (PT) from a predefined set of options. The quality of the embeddings can then be assessed using a probing classifier, with classification accuracy serving as the evaluation metric. The second approach is to directly evaluate the outputs generated from the prompts and then analyze the similarity of the generated text to the original solution. We prefer the first one as the second method introduces risks, since the prompts may leak information already encoded in the embeddings and bias the results. It also relies on identity functions for similarity, e.g., BLEU or ROUGE scores. In addition, prompt engineering techniques can substantially influence the results in a way which is hard to control, as wording in prompts may bias the results to certain programming languages as certain phrases may be used more

Simin Sun and Miroslaw Staron are with the Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, 41756 Gothenburg, Sweden. E-mail: {simin.su, miroslaw.staron}@gu.se.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3629828

often with certain programming tasks/languages. Since we do not know the exact datasets used to train the LLMs, we cannot control for this prompting bias. To avoid these confounding factors, our study focuses on the first approach: evaluating embeddings derived from existing programming repositories, where the PL and PT labels are explicitly defined.

Previously, studies on the ability of LLMs to explain the programs, e.g., as shown by the experiment by Shi et al. [2], indicated the ability of LLMs to explain the source code. The LLMs can provide outlines of the source code, but these explanations are on the level of lines or fragments of code; LLMs do not explain the algorithms implemented in these programs. In this study, our aim is to explore this further and investigate the ability of LLMs to explain programs at higher abstraction levels. We focus on the LLMs internal representation of programs to achieve this goal – the embeddings of the programs in the latent space.

The primary objective of this study is to design and conduct a series of computational experiments to investigate to what extent LLMs support the concept of literate programming, using Rosetta Code [1] and CodeNet Repository [2]. These languages range from highly obscure to simple and easy-to-understand ones. For example, Fig. 2 and Fig. 3 illustrate solutions to the task of calculating the Fibonacci sequence in COBOL and JavaScript, respectively.

The probing classifiers [3], which include a probing task and a probing classifier, are used to evaluate whether a model has learned the desired external properties by analyzing its internal representations. This technique is commonly used to determine whether a pre-trained model embeds and thus captures specific attributes by training a simple classifier on them. The performance of the classifier, usually measured by precision, indicates the effectiveness of the test tasks and provides information on the model's ability to represent the properties targeted [4]. Compared to higher-level evaluations, which focus on the output of LLMs and cannot reveal what the model internally represents, probing classifiers directly test the information encoded in latent embeddings. We adopt probing classifiers, as they provide a measurable way to check whether certain properties can be deduced from the embedding vectors. In our case, this means that if a classifier can accurately predict the programming task or the programming language, then the embedding vectors contain information about the task or the language (or both).

Given the extensive variety of programming languages and tasks, this study aims to assess whether LLMs can comprehend both programming languages and their corresponding descriptions as well. We evaluate multiple models' capabilities by training two classifiers on the hidden state embeddings of programming code using two different labels (programming languages and programming tasks). We analyze these classifiers to examine whether a given program is more closely classified with its counterparts:

[1]https://www.rosettacode.org
[2]https://github.com/IBM/Project_CodeNet

```
Program-ID. Fibonacci-Sequence.
Data Division.
Working-Storage Section.
  01  FIBONACCI-PROCESSING.
    05  FIBONACCI-NUMBER  PIC 9(36)   VALUE 0.
    05  FIB-ONE           PIC 9(36)   VALUE 0.
    05  FIB-TWO           PIC 9(36)   VALUE 1.
  01  DESIRED-COUNT       PIC 9(4).
  01  FORMATTING.
    05  INTERM-RESULT     PIC Z(35)9.
    05  FORMATTED-RESULT  PIC X(36).
    05  FORMATTED-SPACE   PIC x(35).
Procedure Division.
  000-START-PROGRAM.
    Display "What place of the Fibonacci Sequence
    would you like (<173)? " with no advancing.
    Accept DESIRED-COUNT.
    If DESIRED-COUNT is less than 1
      Stop run.
    If DESIRED-COUNT is less than 2
      Move FIBONACCI-NUMBER to INTERM-RESULT
      Move INTERM-RESULT to FORMATTED-RESULT
      Unstring FORMATTED-RESULT delimited by all
    spaces into FORMATTED-SPACE,FORMATTED-RESULT
      Display FORMATTED-RESULT
      Stop run.
    Subtract 1 from DESIRED-COUNT.
    Move FIBONACCI-NUMBER to INTERM-RESULT.
    Move INTERM-RESULT to FORMATTED-RESULT.
    Unstring FORMATTED-RESULT delimited by all
    spaces into FORMATTED-SPACE,FORMATTED-RESULT.
    Display FORMATTED-RESULT.
    Perform 100-COMPUTE-FIBONACCI until
    DESIRED-COUNT = zero.
    Stop run.
  100-COMPUTE-FIBONACCI.
    Compute FIBONACCI-NUMBER = FIB-ONE + FIB-TWO.
    Move FIB-TWO to FIB-ONE.
    Move FIBONACCI-NUMBER to FIB-TWO.
    Subtract 1 from DESIRED-COUNT.
    Move FIBONACCI-NUMBER to INTERM-RESULT.
    Move INTERM-RESULT to FORMATTED-RESULT.
    Unstring FORMATTED-RESULT delimited by all
    spaces into FORMATTED-SPACE,FORMATTED-RESULT.
    Display FORMATTED-RESULT.
```

Fig. 2: Code Implement (Programming Task: Fibonacci-Sequence; Programming Language: COBOL)

```
function fib(n) {
  return n<2?n:fib(n-1)+fib(n-2);
}
```

Fig. 3: Code Implement (Programming Task: Fibonacci-Sequence; Programming Language: JavaScript)

i) in the same programming language (PL) but addressing different tasks (PT), or

ii) in different programming languages (PL) but addressing the same task (PT).

Additionally, we also want to understand whether the hidden state embeddings of programming descriptions capture the same semantics as those of programming code, thereby determining whether LLMs can abstract the content of a given program.

Finally, we explore whether code-specific models, specifi-

cally models fine-tuned on programming code, exhibit similar patterns of recognition and classification as general-purpose LLMs.

The idea of studying this classification stems from the thesis by Bentley and Knuth [5], that when writing programs, one "minimizes the distance between the problem-solving strategies [...] and the program text". In our case, problem-solving strategies are approximated by programming descriptions, and program text is approximated by the source code. Thus, we study the following research question:

RQ1: To what extent do LLMs align with the principles of literate programming?

RQ1.1: Whether the LLMs classify a given program as more similar to its counterparts in i) or ii)?

RQ1.2: Would the LLMs be able to identify the programming task as accurately from the description as from the paired code?

RQ2: How do general-purpose models perform compared to code-specific models in capturing representations? Specifically, do fine-tuned models represent programming languages more effectively than they represent programming tasks?

We conducted computational experiments using 13 general language models and 6 code-specific language models on data from the Rosetta Code and CodeNet repositories[3].

Our results reveal that model size (measured by the number of parameters), maximum input size are critical factors influencing an LLM's ability to understand both code and natural language. Only when the model size reaches the trillion-parameter scale do LLMs effectively achieve literate programming, demonstrating a strong ability to capture the semantics of both natural languages and programming languages. At this scale, models accurately predict both the task and the programming language associated with a given code snippet.

In contrast, models with millions to billions of parameters do not fully realize literate programming. Although increasing the size of the model significantly improves programming language recognition, it does not proportionally enhance the ability to understand the purpose of a given code snippet (task prediction). Similarly, for code-specific models, we did not observe substantial improvement in task prediction as the size of the model increased. However, once a certain size threshold is surpassed, the distinction between general-purpose LLMs and code-specific models diminishes. This is because larger general LLMs also exhibit strong code-recognition capabilities.

The remainder of the paper is structured as follows. Section II explores further the importance of the literate programming for LLMs and the impact of it. Section III summarizes the most recent developments in this area. Section IV explains the methodology used and provides a comprehensive overview of the experimental progress undertaken in our study. Section V presents the results from our experiments. Section VI summarizes the findings, limitations, and future works, and Section VII discusses the validity of our findings. Finally, Section VIII presents our conclusions.

[3]All supplementary data supporting this study are openly available at https://doi.org/10.5281/zenodo.17079423.

II. Motivation

Literate programming, as introduced by Knuth, aims to make programs understandable not only to machines but also to humans by embedding explanations of the design, purpose, and reasoning directly within the code of a higher-level language (called WEB, because it weaves explanations with source code). Such an integration encourages good design practices, makes poor design choices visible, and facilitates comprehension and reuse of complex programs by other developers. Knuth characterized literate programming as a programmer-driven narrative, in which prose and code are intentionally woven together to support human comprehension. In his view, the narrative document is the central artifact, while the source code it produces is secondary. When using LLMs to generate programs, this concepts actualizes again, but in a slightly different form.

In traditional software development, programmers implement software from requirements. Literate programming extends this process by requiring developers not only to write the code but also to provide accompanying descriptions that make the code understandable to others. In the era of LLMs, this dual role is partially delegated to prompts: developers specify requirements in natural language, and the model generates both the code and, in some cases, explanations. Whether an LLM can be considered literate in programming depends on its ability to correctly interpret the requirements expressed in the prompts and internally capture the nuances of different programming languages and programming tasks. This distinction helps explain why most current LLMs excel at programming language recognition but struggle with task recognition, especially when programs are complex. It also sheds light on the inconsistent behaviors often observed in agentic AI systems, such as generating C++ code when C is requested, or shifting languages mid-conversation.

The literate programming via LLMs also depends on the model's ability to recognize and explain existing code. A truly literate model should be able to summarize a given snippet by identifying both the programming task it solves beyond documenting individual lines and the language in which it is written. Such models would be particularly valuable for software engineering tasks that rely on the bridging of code and natural language, such as documentation, code summarization, and educational tools. If a model cannot accurately summarize the code, then it cannot be relied on when generating solutions, i.e., source code, for new problems.

Like traditional software development, which is often difficult for others to understand, LLMs are black-box systems: they generate code without exposing the reasoning process

```
Write a recursive fibonacci implementation in COBOL
```

Fig. 4: Prompt (Programming Task: Fibonacci-Sequence; Instruction: Recursive Implementation; Target Language: COBOL)

that led to their design. Although LLMs can produce code accompanied by explanations, these explanations are typically shallow, focusing on line-level commentary rather than higher-level algorithms and problem-solving strategies. This raises the question of whether LLMs possess the capacity to connect programs with their intended tasks in a manner consistent with the principles of literate programming. Importantly, LLMs do not "understand" inputs in a human sense; instead, they transform all inputs into embeddings within a latent space. The similarity in this space allows the models to group or cluster new content with concepts they have already encountered. In our work, we use the term literate in this embedding-based sense: an LLM can be considered 'literate' if its latent representations allow us to correctly recognize and categorize code according to both the language it is written in and the problem it solves.

This formulation parallels, but also departs from, Knuth's original idea. In the WEB system, semantic alignment between documentation and code was ensured by the weaving and tangling process explicitly authored by humans. In contrast, for LLMs, this equivalence must be inferred from the alignment of natural language and code in their latent space. Studying this alignment provides a principled way to evaluate whether LLMs move beyond code generation toward the literate programming ideal. Thus, while our hypothesis captures an essential dimension of Knuth's vision: maintaining equivalence between code and explanation, it substantially reframes literate programming in the context of LLMs. Therefore, we clarify that our goal is not to replicate Knuth's original framework, but to extend its underlying principles to assess whether LLMs can achieve literate behavior through aligned representations of code and natural language.

A. Importance

Literate programming is crucial for LLMs because these models serve as the foundation for systems like ChatGPT and GitHub Copilot. Programmers depend on these tools to perform specific tasks in the appropriate programming language. If LLMs struggle to differentiate between tasks associated with different programming languages, programmers cannot trust the quality of their outputs. This dependence on LLMs can jeopardize the quality of the generated software. For instance, consider the prompt in Figure 4, which includes both the task description and the programming language specification.

The Fibonacci sequence is a well-known mathematical concept, making it relatively easy to find reference implementations. The details are essential: the recursive implementation and COBOL. We expect the LLM to provide us with the correct implementation, which it often does. However, sometimes LLMs can answer in another programming language (e.g., Python) or generate iterative calculations, rather than recursive ones. For this prompt, it is easy to check the correctness, but we want to use LLMs as assistants for much more advanced tasks, where we do not have an oracle yet – solving real programming issues in the software engineering industry. Our focus lies on more complex tasks that require advanced capabilities, such as Mixture-of-Experts models, to solve them. This means that we need to depend on the models' ability to differentiate between various tasks. So, *literate LLMs* can provide the correct solution to the problem in the correct programming language. In contrast, *illiterate LLMs* can give a solution to a different problem or in a different programming language. If an LLM is illiterate, then we cannot use it directly to generate solutions, but we need additional technologies, like RAGs (Reality Augmented Generation) to provide the programmers with reliable solutions to advanced programming problems.

III. Related Work

We begin by exploring research and practices in the domain of literate programming. Next, we examine studies on evaluating LLMs, including their hypotheses and research methods. Since how code-specific models enhance performance is also part of the research scope, we review the development of these models. Finally, we study research on code-related tasks that could benefit from our study.

A. Literate Programming

Literate programming was introduced as an idea by Knuth [1] and further elaborated in [5]. This concept asserts that programming should be presented as a combination of explanations in natural language and source code, making it comprehensible both to human readers and to computers. In the early 2010s, Org-mode [6] was one of the initial attempts to integrate this programming paradigm into the Emacs text editor. The idea gained significant attention during the 2010s, with the advent of notebook interfaces. These tools, which facilitated literate programming, became increasingly popular among programmers, particularly within the field of data science [7]. Codestrates [8] is one such example that offers collaborative and interactive notebook environments.

With the rise of generative AI, the implementation of literate programming has taken new dimensions. As technology has rapidly evolved, the concept of code literacy has gained greater importance, especially since natural language interfaces are used for programming [9]. The development of LLMs has further expanded the scope of literate programming. Many recent studies propose leveraging LLMs to generate natural-language summaries of code snippets, echoing the principles of literate programming. For instance, Natural Language Outlines (NL outlines) [2] is a tool designed to divide code into logical segments and provide natural language summaries for each section. This represents a new dimension of literate programming, focusing on a fine-grained assessment of code. By focusing on small code snippets, such tools excel

at explaining localized functionality, but may overlook the broader, abstract structure of the program, i.e., design patterns or the task that is solved. As the interplay between natural language and programming continues to evolve, these tools highlight the shifting boundaries of what literate programming can achieve in the era of generative AI. These works highlight that literate programming has always been about aligning code with explanation. Our study builds on this tradition but shifts the focus: instead of humans deliberately weaving code and prose, we investigate whether LLMs internally represent this alignment. This motivates our "literate programming hypothesis for LLMs."

B. Evaluating LLMs

The evaluation of LLMs in software engineering has attracted significant attention, with more than 1,000 publications in 2023 alone [10]. Although much of this work uses higher-level evaluations to assess downstream performance (e.g., code completion, generation), fewer studies examine what LLMs internally represent.

Early research on the naturalness of software provides one perspective. Hindle et al. [11] showed that source code, like natural language, exhibits statistical regularities that can be captured by simple n-gram models. However, such studies focus on surface-level predictability and do not assess whether models capture higher-level semantics or task-level features. In our framing, naturalness reflects local regularities, whereas literate programming requires models to align the code with its underlying problem-solving intent.

A more direct method for evaluating internal representations is probing classifiers, originally developed in NLP to test whether embeddings capture linguistic properties. Probing assesses pretrained models by analyzing the extent to which they encode specific linguistic properties. Early studies [12], [13] focused mainly on probing word embeddings, examining properties such as referential knowledge, morphology, and syntax. Subsequent research expanded to sentence embeddings, exploring their ability to encode syntactic structures [14] and semantic properties [15]. With the rise of code-specific models, probing techniques were first applied to source code by Karmakar and Robbes [4]. They evaluated four probing tasks using four different pretrained models to analyze how well these models capture various code properties. Later, Troshin and Chirkova [16] extended this work by examining the representations generated by multiple pretrained models in eight diagnostic tests. Their findings suggest that, while these models effectively capture surface-level information, such as syntax, identifier usage, and namespace structures—they struggle with more complex properties, such as semantic equivalence. Additionally, they observed that training on code-specific data significantly improves the models' ability to understand code constructs. Ahmed et al. [17] employ different methods from probing to investigate LLMs' semantic 'understanding'. Instead of evaluating the representations obtained for subsequent tasks, they directly explore the representations acquired by the language models. They employ two identical operators, operand swap and block swap, which alter the code structure without altering its semantics. By masking key elements of these alterations, they task the models with predicting the masked segments. All models utilized in the study successfully predict the masked operators, enabling a direct assessment of the LLMs recognition of code regarding format changes. However, the study's scope is limited as the operators employed are simplistic and do not substantially alter the logic or variables within the code. However, Chen et al. [18] showed that generating domain models in textual forms is still challenging for LLMs. Even models like GPT-3.5 and GPT-4 have low accuracy in generating relationships (0.34) and moderate accuracy in generating classes (0.76) and attributes (0.61).

Our work builds on this line of research, but diverges in focus. Rather than probing for syntax, identifiers, or simple structural changes, we evaluate whether LLM embeddings capture programming languages and programming tasks, two properties central to our definition of literate programming for LLMs. Specifically, we use probing classifiers to test whether a given program is more closely aligned with other solutions written in the same language or with those solving the same task (RQ1.1), and whether task information is equally well represented in code and its natural language description (RQ1.2). In this way, we extend the probing from surface-level code properties to the higher-level semantic alignment that underlies literate programming.

LLMs for Code: Transformer-based models such as GPT and BERT are widely used in natural language processing tasks such as sentiment classification, text generation, language translation, question answering, and summarization. Over time, as these models have evolved in architecture and size, their performance in linguistic tasks has improved significantly. However, concerns have arisen about how well pretrained models can capture semantics of the source code. While most models are designed to process natural language, some are specifically developed to focus on capturing code semantics.

Kanade et al. [19] introduced CuBERT, the first attempt to train the BERT [20] model on source code. They trained the model using Python code collected from GitHub, significantly improving performance in five code-related tasks, including Variable Misuse detection. Similarly, Feng et al. [21] developed CodeBERT, an extension of BERT trained on six programming languages: Go, Java, JavaScript, PHP, Python, and Ruby. CodeBERT functions as a bimodal model capable of processing both natural and programming language structures. Karampatsis and Sutton [22] proposed SCELMo, a model trained on ELMo [23] for bug detection. Their work highlights the benefits of bidirectional token embeddings in providing additional context for improved performance. Guo et al. [24] introduced GraphCodeBERT, which enhances pre-training by incorporating data flow instead of traditional representations, capturing a deeper structural recognition of code. This approach leads to significant performance gains in various downstream tasks.

These code-specific models highlight how domain-focused pretraining enhances the capture of structural and semantic code features. However, they are primarily evaluated through downstream tasks, such as defect detection or code completion. In contrast, our study investigates whether such models, compared to general-purpose LLMs, better align code with its underlying programming tasks and languages. This directly addresses RQ2, which asks whether fine-tuned code-specific models represent programming languages more effectively than they represent programming tasks. By framing this evaluation through the lens of literate programming, we move beyond performance on isolated tasks and instead assess whether these models internally encode the semantic alignment between code and its explanation.

Code-related Tasks: We aim to explore literate programming through LLM due to the growing popularity of LLM-based tools for code-related tasks, such as code generation and summarization. Our goal is to determine which models can effectively solve these tasks while inhibiting a better recognition of programming tasks and codes. These tasks rely heavily on how well the models align code and natural language. A better "literate programming" ability would directly improve summarization (code→description), generation (description→code) and translation (interlanguage task recognition). Our probing experiments thus provide foundational evidence for why LLMs succeed or fail at such tasks.

Code generation involves converting natural language descriptions or keywords into executable code. A significant amount of research in this field has led to commercial tools like Copilot, CodeAI, StarCode, and DeepCode. Early efforts [25], [26] focused on translating natural languages into an executable, web-hosted or domain-specific languages. These approaches were primarily token-based and struggled with recognizing complex descriptions. Later advancements utilized Recurrent Neural Networks (RNN) [27] and reinforcement learning [28] to enhance performance for various purposes. Lin et al. [27] used their model to generate shell commands, while [28] introduced SEQ2SQL to generate SQL commands. More recently, deep learning-based approaches have emerged [29], [30]. Le et al. [29] extended CodeT5 with deep reinforcement learning, improving pretraining, processing, and improving performance on relevant benchmarks. Svyatkovskiy et al. [30] proposed IntelliCode Compose, a multilingual code completion tool. They further trained GPT on source codes in four programming languages: Python, C#, JavaScript, and TypeScript.

Code summarization and code generation represent opposite processes: the former translates source code into natural language, while the latter converts natural language into source code. According to Zhang et al., [31], code summarization approaches can be categorized into four according to how they represent the source code: token-based, tree-based, graph-based, and others. Furthermore, based on the techniques employed, the code summarization can also be classified into seq2seq [32], [33], RNN [34], [35], Bi-LSTM [36], [37], and transformer [21], [38] based, etc.

Code translation, also called source code to source code translation, involves the conversion of high-level programming languages into another form. For instance, in 2015, Aggarwal et al. [39] expanded the capabilities of Moses [40] to facilitate the translation from Python 2 to Python 3. Similarly, as demonstrated in [41], there are tools that can convert CoffeeScript to JavaScript. In particular, TransCoder [42] represents an unsupervised learning model adept at translating between C++ and Java.

IV. Research design

A. Experiments

In this paper, we investigate whether LLMs achieve literate programming, defined as the ability to recognize both PL and PT equally well from a given program. This definition connects directly to the principles of literate programming: code and its explanation must be semantically aligned. If an LLM embedding encodes enough information to accurately distinguish languages and tasks, this suggests that the model is 'literate' in the sense of simultaneously capturing syntactic and semantic properties.

To evaluate this, we designed four complementary experiments. Each experiment investigates a different aspect of literate programming and contributes evidence to answer RQ1 and RQ2. Our design follows three steps: (i) preprocess and sample Rosetta Code data (and later validate on CodeNet), (ii) extract code and text embeddings from each LLM, and (iii) train probing classifiers to test whether embeddings capture PL and PT distinctions. Fig. 5 illustrates the overall experimental design.

- Experiment 1: **General-purpose models for all languages and all tasks.** Extract embedding representations of code from various LLMs and use the corresponding programming languages and tasks as labels. Train two separate classifiers to determine whether a given program is more closely associated with other programs written in the same programming language but for different tasks or with programs written in different languages but for the same task.
- Experiment 2: **General-purpose models but a limited and balanced number of languages and tasks.** Construct a balanced subset of programming languages and tasks, selecting sets of 5, 10, 20, 30, 40, and 50 languages and tasks. Compare the classification performance across these subsets with the results of Experiment 1 to assess how the number of languages and tasks affects model performance.
- Experiment 3: **General-purpose models predict the task descriptions using the PT classifier.** Use the programming task classifier trained in Experiment 1 to predict embedding representations for the task description.
- Experiment 4: **Language model trained on code.**. Repeat Experiments 1, 2, and 3 using a set of code-specific models.

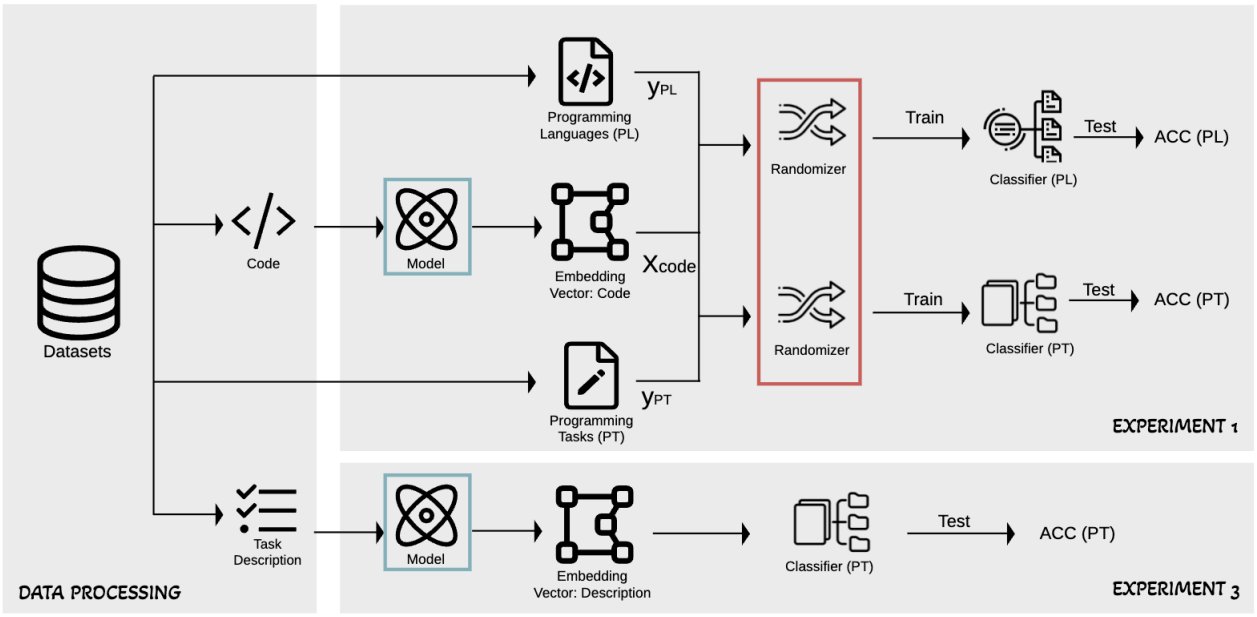


Fig. 5: Overview of Data Preparation and Hypothesis Evaluation. This process involves training classifiers (PL and PT) for prediction in Experiment 1. In Experiment 2, a randomizer (red box) was added to choose a limited but balanced number of records from the dataset. In Experiment 3, we assess the semantic alignment between code and natural language descriptions using the PT classifier. In Experiment 4, we change the model to code-specific models, as marked in blue.

Experiment 1 investigated whether LLMs, using the same code embedding representations, are better at identifying the programming task that a code snippet is intended to solve or the programming language in which it is written. We train two classifiers: one to predict the programming language from code embeddings, and another to predict the programming task. This setup directly tests whether embeddings encode syntactic information (PL) or semantic intent (PT), and how strongly. The comparison between PL and PT accuracy is not about one being "better" or "worse," but about revealing the relative strength of syntax vs. semantics in LLM embeddings. A model that succeeds in PL but fails at PT may rely on surface-level features rather than a deeper problem-solving alignment. This experiment directly addresses RQ1.1.

Experiment 2 further examined the impact of corpus size on model performance for programming languages and tasks. We repeat Experiment 1 with balanced subsets of 5, 10, 20, 30, 40 and 50 tasks / languages. This controls for skew in Rosetta Code (many languages with few examples each) and tests how the scale and balance of the data set affect the discriminative power of embeddings. This design allows us to disentangle whether the performance differences are due to the capacity of the model or the data distribution. It provides a more rigorous test of RQ1.1.

Experiment 3 reused the PT classifier from Experiment 1 to classify embeddings of textual task descriptions. If performance remains high, this indicates that the embeddings for code and descriptions are semantically aligned, a core criterion of literate programming. If performance collapses, it suggests that the models rely mainly on syntactic cues. This experiment evaluates RQ1.2, testing whether models represent code and natural language tasks consistently.

Experiment 4 repeated Experiments 1 and 3 using models pretrained specifically on code. This tests whether domain-specific pretraining improves task alignment or simply reinforces syntactic pattern recognition. By comparing general-purpose and code-specific LLMs, we address RQ2, asking whether specialized training enhances literate programming ability.

Together, these experiments form a coherent framework: Experiment 1 contrasts syntax and semantics; Experiment 2 controls for dataset scale; Experiment 3 connects code and natural language; and Experiment 4 compares general-purpose and code-specific models. This layered design ensures that our results speak directly to the literate programming hypothesis, while also uncovering the conditions under which LLMs succeed or fail to achieve it.

The foundation of our experimental design lies in analyzing the embedding representations of programming languages and natural language descriptions. Given the training dataset $\{X_{train\_code}, y_{train\_PT}, y_{train\_PL}\}$ and test data $\{X_{test\_code}, X_{test\_text}, y_{test\_PT}, y_{test\_PL}\}$, we explore their representations by using them for classification. The embedding can therefore be written as Equation $E_{train\_code} = M(X_{train\_code})$, where M stands for the model we used, and E_{train_code} represents the embeddings obtained from the last hidden layer of the model. Using these embeddings, we train two separate classifiers: one using $(X_{train}, y_{train\_PT})$ to classify programming tasks, and the other using $(X_{train}, y_{train\_PL})$ to classify programming languages. The performance of these classifiers is evaluated by

comparing their predicted labels with the ground truth, yielding accuracy scores ACC_{PT} and ACC_{PL}, respectively. Additionally, to assess whether the embeddings of code capture semantic similarities with their corresponding natural language descriptions, we apply the classifier trained on (X_{train_code}, y_{train_PT}) to predict labels for text embeddings X_{test_text} and get the accuracy scores ACC_{text}. This enables us to determine whether LLMs embed programming concepts in a way that aligns with human-readable task descriptions.

We hypothesize that the population mean of the test accuracy for classifiers trained in programming languages is approximately equal to that of classifiers trained on programming tasks. Furthermore, both should be approximately equal to the accuracy of predicting the textual descriptions, as formulated in Equation 1. This formulation allows us to test whether LLMs achieve a unified representation of code and text, supporting the principles of literate programming.

$$\mu_{ACC_{PL}} = \mu_{ACC_{PT}} \tag{1a}$$

$$\mu_{ACC_{text}} = \mu_{ACC_{PT}} \tag{1b}$$

B. Datasets

We reviewed several existing datasets for code and programming tasks, including HumanEval [43], HumanEval-ET [44], HumanEval-XL [45], MBPP [46], MBPP-ET [44], CodeNet [47], PIE [48], Transcoder [42], CodeSearchNet [49], and Rosetta Code [50]. A summary of these datasets is provided in Table I.

To ensure fair comparisons, we established two main criteria for selecting datasets: (1) the number of programming tasks and programming languages should both be sufficiently large and roughly balanced after processing, and (2) each programming language and task should contain at least two records to allow for a train-test split.

TABLE I: Statistics of Surveyed Datasets

Dataset	#P_Language	#P_Task	Total
HumanEval	1	164	164
HumanEval-ET	1	164	164
MBPP	1	974	∼1K
MBPP-ET	1	974	∼1K
PIE	1	2,530	∼80K
Transcoder	3	524	∼3M
CodeSearchNet	6	2M	∼6M
HumanEval-XL	12	820	∼22K
CodeNet	55	4,053	∼14M
Rosetta Code	926	1,228	114K

Based on these criteria, only CodeNet and Rosetta Code can meet the requirements. Rosetta Code [4] aims to gather solutions for the same programming task implemented in various programming languages. As of March 2024, it contains 1,228 tasks with implementations in 926 languages; any submissions after this date were excluded. In contrast, CodeNet [5], introduced by IBM as part of Project, and it includes approximately 4,000 programming problems, each associated with multiple solutions implemented in various programming languages. In total, the dataset contains 55 programming languages.

[4]https://github.com/acmeism/RosettaCodeData

[5]https://github.com/IBM/Project_CodeNet

TABLE II: Statistics of the CodeNet and Rosetta Code (Rosetta) Datasets and Processed Data

Name	Rosetta	CodeNet
# Programming Tasks (PT)	1,228	1,406
# Programming Languages (PL)	926	55
# PT * PL Combinations	83,997	11,164
# Total Records	114,376	11,998,889
After processing		
# Programming Tasks (PT)	532	55
# Programming Languages (PL)	525	52
# Records over all PT * PL Combinations (if $\#Records_{per\_combination} \geq 1$)	38,677	1,727,817
# Train	24,892	1,725,993
# Test	13,786	1,824

In this study, we use Rosetta Code as the primary experimental dataset and CodeNet for validation. This choice is motivated by two considerations. First, Rosetta Code provides greater diversity in both programming languages and tasks, making it particularly suitable for evaluating the ability of LLMs to recognize the language and task of given code snippets. Second, Rosetta Code contains fewer total records than CodeNet, and its smaller scale enables the training of more classifiers within feasible computational limits.

However, additional data processing is still necessary to further filter the data and ensure a fair classification. Both datasets provide information on the programming description (text), the source code (code), and their associated languages (PL) and tasks (PT). We restructure each record into the format ($\mathbf{Code}, \mathbf{Text}, \mathbf{PL}, \mathbf{PT}$), where ($\mathbf{Code}, \mathbf{Text}$) represent the input data and ($\mathbf{PL}, \mathbf{PT}$) serve as the corresponding labels.

C. Data Preparation

Table II presents more details about both datasets, including the count of tasks and languages. Note that each combination of programming languages and tasks may have multiple solutions, while some have none. Considering the task of calculating the Fibonacci sequence as an example, there are a total of 445 solutions available for this task in the filtered train dataset.

Data processing involved obtaining the necessary data and consolidating it into a single comma-separated text file for analysis. Given the substantial data in the original dataset, we implemented a filtering mechanism based on the combination of programming language and task. If multiple records were found, we retained the data and partitioned it. One record was designated as the test data, while the remaining were allocated as training data. This procedure ensured adequate data availability for each programming language and task combination, facilitating accurate similarity assessment during calculations.

As indicated in Table II, 38,677 Rosetta Code records and 1,727,817 CodeNet records meet the specified criteria. To further understand the dataset, we analyzed the length of the

TABLE III: Summary Statistics of Line and Token Counts per Record

Line			Token		
Stat	Rosetta	CodeNet	Stat	Rosetta	CodeNet
mean	17.76	41.36	mean	68.42	397.59
std	31.78	130.61	std	142.46	4,893.27
min	2	1	min	422.18	146
50%	8	25	90%	375	517
75%	19	42	95%	590	811
max	1,033	24,277	max	8,506	379,407

records to determine how many could fit within the model's maximum input length, informing our truncation and padding strategy. The results, presented in Table III, indicate that 95% of the data consists of fewer than 517 tokens.

D. Embeddings

After processing the data, we extract embeddings for each record – code and textual description – using 19 different models. The statistics for these models are detailed in Table IV.

These models can be categorized in three ways:

- By Purpose: Models are classified as either general-purpose language models or those specifically designed for code summary and generation.
- By Model Type: The models fall into two broad categories: pretrained language models (PLMs), such as BERT and RoBERTa, and LLMs. Although both are transformer-based models, as model sizes increase, traditional PLMs such as BERT are often not considered LLMs. However, since there is no strict definition of what constitutes an LLM in terms of parameter count, we categorize such models as PLMs for clarity.
- By Scale: Based on the number of parameters, the models are grouped into three tiers: million-scale models, billion-scale models, and trillion-scale models, as outlined below.
  - Million-scale models:
    - BERT [20]: was utilized in the base uncased version with 110M parameters.
    - GPT2 [51]: was used in its smallest version with 124M parameters.
    - RoBERTa [52]: was utilized in its base version, comprising 125M parameters and case sensitive.
    - CodeBERT [21]: the *codebert-base* model from Hugging Face, trained on bimodal data covering six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go. We expected it to improve accuracy compared to BERT.
    - CodeGPT [53]: the *CodeGPT-Multilingual* [6] model from Hugging Face, trained by Nadine Kuo on source code in Java, Python, C++, Kotlin, Go, and Julia. This choice was preferred over the official CodeGPT2 released by Microsoft, which only focuses on one programming language (Java or Python), as it may not suffice compared to the multilingual approach of CodeBERT.
  - Billion-scale models:
    - Falcon [54]: The architecture of this model is based on GPT3, with a few notable distinctions. Specifically, we utilize the version with 7 billion parameters *tiiuae/falcon-7b* and 40 billion parameters *falcon-40b* released in 2023 and their latest model *falcon-11b* released in 2024 with extended input size from 2048 to 8192.
    - LLaMA-2 [55]: This model was developed by Meta and has three sizes; we evaluate *LLaMA-2-7B-hf* and *LLaMA-2-13B-hf*.
    - LLaMA-3.1 [56]: This is a newer version of the LLaMA models that launched in 2024, and we included two sizes: *LLaMA-3.1-8B* and *LLaMA-3.1-70B*.
  - Trillion-scale models:
    - Text-Embedding-3 [7]: OpenAIs latest and most potent embedding model. We include three models: *Text-embedding-3-ada-002*, *Text-embedding-3-small* and *Text-embedding-3-large*. They have the same maximum input sizes, but differ in the output dimensions.

While getting the embeddings from the pretrained models, inputs are truncated based on the maximum input length supported by the models, without applying padding. This decision is justified by the observation that 90% of the records fit within the input size limit of the smallest model used, BERT, which has a maximum input length of 512 tokens. Furthermore, the average input length for all records is only 68.42 and 397.59 tokens, as shown in Table III. Padding, even with masking, would lead to inefficient computation by requiring the attention mechanism to process many unnecessary padded tokens. To optimize resource usage, we chose to truncate inputs that exceed the maximum length instead of padding shorter ones.

The extracted embeddings are the last hidden states, and then they are averaged across the sequence dimension, expressed as $last\_hidden\_state.mean(dim = 1)$. We deliberately avoided using a pooler layer, since several models in our study (e.g., GPT-2, CodeGPT2) do not provide one, and relying on it would introduce inconsistencies. Averaging across the sequence dimension ensures comparability across architectures and has been shown to yield outcomes similar to those obtained from pooler layers. We also considered whether intermediate layers might offer semantically richer embeddings. However, our study spans a variety of architectures with different depths and training strategies; selecting "middle layers" would introduce arbitrariness and potential bias into the

[6] http://resolver.tudelft.nl/uuid:2b2386e8-f9a9-4d77-9a4a-a4e7a5208d38

[7] https://platform.openai.com/docs/guides/embeddings/

TABLE IV: Statistics of Chosen Models

	Model Name	Release Time	# Paramter	Max. Input	Output Size
Million-scale models	BERT	2018	110M	512	768
	GPT2	2019	124M	1024	768
	RoBERTa	2019	125M	512	768
	CodeBERT	2020	125M	512	768
	CodeGPT2	2023	124M	1024	768
Billion-scale models	LLaMA-2-7B	2023	7B	4096	4096
	LLaMA-3.1-8B	2024	8B	131072	4096
	LLaMA-2-13B	2023	13B	4096	5120
	LLaMA-3.1-70B	2024	70B	131072	8192
	Falon7b	2023	7B	2048	4544
	Falcon11b	2024	11B	8192	4096
	Falcon40b	2023	40B	2048	8192
	CodeLLaMa7b	2023	7B	16384	4096
	CodeLLaMa13b	2023	13B	16384	5120
	CodeLLaMa34b	2023	34B	16384	8192
	CodeLLaMa70b	2023	70B	16384	8192
Trillion-scale models	Text-embedding-3-ada-002	2022	∼trillion*	8196	1536
	Text-embedding-3-small	2024	∼trillion*	8196	1536
	Text-embedding-3-large	2024	∼trillion*	8196	3072

*:Estimated number, the exact number of parameters is not revealed in public documentation to prevent the disclosure of internal architectural details.

analysis. In contrast, the last hidden state represents the final internal representation of the model before prediction, and averaging across the sequence dimension produces embeddings that are both general-purpose and architecture-agnostic. This makes last-layer mean pooling the fairest and widely adopted approach to cross-model probing. Another alternative strategy is to use the embedding of the special classification token (e.g., [CLS] in BERT-like models). However, not all architectures in our study implement such a token (e.g., GPT2, CodeLLaMA), making it unsuitable for cross-model comparison. Moreover, recent studies [57] have shown that the mean pooling of the last hidden states can achieve equal or better performance than relying solely on the [CLS] embedding, as it captures information distributed across the entire sequence rather than being tied to a single position. For these reasons, we adopt the last hidden state averaging as the most fair and robust approach for probing across diverse model families.

To visualize the embeddings we obtained and highlight their features, we utilized t-SNE (t-Distributed Stochastic Neighbor Embedding) [58], a widely used technique for reducing the dimensionality of high-dimensional data and representing it in a lower-dimensional space.

E. Classification

Following the extraction of the embeddings from a model, we used them to train two distinct classifiers with two sets of labels: one representing the programming languages of the code and the other representing the programming tasks associated with the code. After training both models, we evaluated their performance on the test data, reporting the accuracies as ACC_{PL} and ACC_{PT}. Additionally, we assessed the quality of task description embeddings by using the model trained on programming tasks to classify task descriptions, denoted as ACC_{text}. If the code embeddings capture sufficient information, they should enable effective differentiation between task descriptions.

To evaluate the embeddings, we explored three classifiers: k-nearest neighbors algorithm (KNN), support vector machines (SVM), and Convolutional Neural Network (CNN). The selection criteria for these classifiers were two-fold. First, a probing classifier should be a simple model without hidden layers, which serves as an essential diagnostic tool [4]. By keeping the classifier simple and largely free of hidden layers, we ensure that performance reflects the quality of the embeddings themselves, rather than the classifier's capacity to learn complex decision boundaries. For this reason, we excluded sophisticated architectures such as LSTMs or BERT, which could mask weaknesses in the embeddings by compensating with their own representational power. Second, the classifiers needed to handle high-dimensional feature spaces: our embeddings reach up to 8,192 dimensions, with labels spanning over 500 categories. This ruled out models such as Naïve Bayes, which are not well-suited for such conditions.

For CNN, we deliberately adopted a minimalistic architecture consisting of two convolutional layers, each followed by ReLU activation and max pooling. Although this design constrains the absolute performance of the network, it aligns with our diagnostic objective: to test whether the embeddings alone contain discriminative features for the programming language and task classification. Using a deeper or more expressive CNN could have improved accuracy, but at the cost of confounding whether performance gains arose from the embeddings or from the classifier itself (overfitting). In our design, the convolutional layers extract meaningful features, while max pooling reduces spatial dimensions to mitigate overfitting and improve computational efficiency. The fully connected layers then map the extracted features to the final classification output, with fully connected layers 1 containing 128 hidden units and fully connected layers 2 having num_classes neurons for classification. We set the kernel size to 3, padding to 1, and max pooling to 2, balancing feature extraction with dimensionality reduction while using ReLU activations to introduce non-linearity. To maintain a controlled

evaluation, we only fine-tuned the learning rate and batch size. We also excluded computationally expensive classifiers such as random forests, since the combination of very high-dimensional embeddings and hundreds of labels would have required prohibitive training resources.

F. Computational Resource

To obtain embeddings from different LLMs, we employed a range of GPU configurations:

- NVIDIA A40: 48 GB VRAM, 64 GB system memory per GPU.
- NVIDIA A100: 40 GB VRAM, 128 GB system memory per GPU.
- NVIDIA A100: 80 GB VRAM, 256 GB system memory per GPU.

For classifier training, we used different setups depending on the model type. KNN and SVM classifiers were trained on a 16-core CPU with 256 GB system memory. For CNN classifiers, we used an NVIDIA V100 with 32 GB VRAM and 384 GB system memory per GPU.

V. Results

A. Experiment 1

General-purpose models for all languages and all tasks: In this experiment, we analyze the embeddings extracted from 13 models by training two classifiers and evaluating their accuracy in predicting PL and PT. The results are presented in Table V. Our analysis demonstrates that trillion-scale models, particularly the Text-embedding-3 series, excel in predicting PT, while the remaining models achieve higher accuracy in predicting PL. Million-scale models, such as BERT, GPT-2, and RoBERTa, generally underperform compared to more recent billion-scale models like LLaMA and Falcon.

There is a notable accuracy gap between the predictions of PL and PT, ranging from approximately 0.15 to 0.5 for most models. This gap is less noticeable under the SVM classifier. For example, LLaMA-3.1-8B achieved 0.8552 accuracy for PL versus 0.7410 for PT with SVM, a gap of 0.11. In contrast, CNN exhibited much larger disparities, frequently exceeding 0.3, as it performed reasonably well on PL classification but consistently struggled with PT prediction.

As the size of the model increases, overall performance improves in both tasks, with larger models consistently outperforming smaller ones. However, LLaMA-3.1-8B surpasses the larger but older LLaMA-2-13B. In particular, LLaMA-3.1-8B has a longer maximum input size and a shorter output size compared to LLaMA-2-13B, suggesting that factors beyond model size, such as maximum input length and architecture, also influence performance.

Another exception is observed in the Text-embedding-3 series, which achieves consistently high accuracy across both tasks and ranks among the top three models for PL prediction (as highlighted in bold in the table). Interestingly, despite their strength in PT prediction, these models also perform competitively in PL classification, sometimes ranking among the top three models and consistently narrowing the performance gap between PL and PT. Referring to our design in Equation 1, this suggests a well-balanced capability across both tasks and indicates progress toward achieving literate programming.

Looking at the three classifiers we compared, we notice that SVM performs the best as it shows the highest accuracy across all tasks and all models. Their ability to interpret high-dimensional vector representation is better compared to KNN and the simple CNN we are using. The CNN architecture we are using might require further investigation or a fine-tuning process, as it did well in predicting the programming languages but not in the other two classification tasks.

B. Experiment 2

All models but a limited and balanced number of languages and tasks: As demonstrated in Experiment 1 (Table V), the CNN struggled with the PT tasks (e.g., for LLaMA-3.1-70B, PL $\approx$ 0.82 vs. PT $\approx$ 0.01) and relied heavily on the size of the training data. Therefore, we did not include it in these experiments. Tables VI and VII present the median accuracy results from ten random trials using KNN and SVM classifiers across different sample sizes.

Across models and sample sizes, SVM yields higher accuracy, lower variance, and more stable behavior than KNN for both PL and PT. For example, at size 10, LLaMA-3.1-8B achieves 0.9293 (PL) and 0.8916 (PT) with SVM (Table VII), compared to 0.7856 (PL) and 0.5598 (PT) with KNN (Table VI). This pattern holds broadly, indicating the advantage of SVM in finding margins in high-dimensional embedding spaces. Consequently, the analyses below emphasize the SVM results.

With SVM, we observe mild degradation as the class set grows (5$\rightarrow$50), particularly for older/smaller models. For example, for RoBERTa, the accuracy of PT decreases from 0.7998 (size 5) to 0.7095 (size 50); BERT PT decreases from 0.8670 to 0.7423 (Table VII). This is expected: increasing the number of classes raises the complexity of the decision while holding per-class evidence fixed. In contrast, stronger models (e.g., LLaMA/CodeLLaMA, Text-embedding-3) remain robust, showing smaller declines.

- Million-Scale Models (BERT, GPT-2, RoBERTa): These models generally show a decreasing trend in both PL and PT tasks as the sample size increases from 5 to 50. They also consistently underperform compared to modern, specialized architectures such as LLaMA and CodeLLaMA, highlighting the limitations of smaller-scale models in capturing programming contexts.
- Billion-Scale Models: LLaMA-3.1-8B/70B sustain strong PL with SVM across sizes (e.g., 8B: 0.9652$\rightarrow$0.8891 PL, 0.9232$\rightarrow$0.8621 PT from size 5$\rightarrow$50), while Falcon-11B/40B are competitive but slightly weaker on PT.

TABLE V: Experiment 1 Results: PL and PT Classification Accuracies Using Embeddings from General-Purpose Language Models. Bold text highlight the top 3 highest accuracy for each classification task.

	Classifier	KNN			SVM			CNN		
	Accuracy	PL	PT	text	PL	PT	text	PL	PT	text
General-purpose Models	BERT	0.4544	0.2929	0.0583	0.7043	0.5641	0.1992	0.6029	0.3619	0.1387
	GPT2	0.2138	0.0538	0.0056	0.4904	0.1504	0.0169	0.6746	0.2532	0.1172
	RoBERTa	0.4288	0.1843	0.0244	0.7204	0.4779	0.1053	0.6303	0.3128	0.1230
	Falon7b	0.5717	0.1873	0.0376	0.8240	0.7069	0.3571	0.7163	0.1479	0.0508
	Falon11b	0.6481	0.2658	0.0489	0.8292	0.7249	0.3797	0.7118	0.0117	0.0020
	Falon40b	0.6435	0.2418	0.0395	0.8398	0.7249	0.3177	0.3234	0.0046	0.0020
	LLaMA-2-7B	0.6468	0.2503	0.0771	0.8313	0.7149	0.4323	0.7612	0.4390	0.1953
	LLaMA-3.1-8B	**0.7239**	0.2726	0.1673	0.8552	0.7410	0.5338	0.7878	0.4770	0.2578
	LLaMA-2-13B	0.7063	0.2682	0.0940	0.8495	0.7385	0.4549	0.7851	0.4893	0.2617
	LLaMA-3.1-70B	**0.7423**	0.2458	0.0714	**0.8648**	0.7452	0.4342	**0.8157**	0.0129	0.0020
	Text-embedding-3-ada-002	0.5711	**0.6956**	**0.8722**	0.8453	**0.8524**	**0.9361**	0.6946	**0.6533**	**0.7305**
	Text-embedding-3-small	0.4808	**0.6928**	**0.8346**	0.8122	**0.8242**	**0.8929**	0.6515	**0.5527**	**0.6113**
	Text-embedding-3-large	0.5473	**0.7661**	**0.9248**	0.8544	**0.8703**	**0.9624**	0.7331	**0.5967**	**0.6914**
Models for Code	CodeBERT	0.4357	0.1465	0.0169	0.7149	0.4019	0.0733	0.5468	0.1531	0.0371
	CodeGPT2	0.5344	0.2958	0.0414	0.7750	0.6401	0.2688	0.6801	0.4178	0.2012
	CodeLLaMa-7B	0.7141	0.2797	0.0714	0.8566	0.7562	0.4699	0.8018	0.5103	0.2422
	CodeLLaMa-13B	**0.7245**	0.2782	0.0752	0.8642	0.7621	0.4887	0.7991	0.3013	0.1934
	CodeLLaMa-34B	**0.7249**	0.2725	0.0470	**0.8664**	0.7634	0.3910	**0.8293**	0.0442	0.0098
	CodeLLaMa-70B	0.7081	0.2581	0.0395	**0.8660**	0.7543	0.3910	**0.8230**	0.2184	0.0918

TABLE VI: Results of Experiment 2: Median Accuracy from Ten Random Selections across Six Sample Sizes Using KNN. Bold text highlights the top 3 highest accuracy for each classification task.

KNN	5		10		20		30		40		50	
Model	PL	PT	PL	PT	PL	PT	PL	PT	PL	PT	PL	PT
BERT	0.5269	0.5100	0.5283	0.4991	0.5143	0.4811	0.4715	0.4821	0.4887	0.4928	0.4830	0.4805
GPT2	0.4303	0.3797	0.2910	0.2571	0.2586	0.1956	0.2205	0.1777	0.2184	0.1682	0.2067	0.1648
RoBERTa	0.4836	0.4744	0.4299	0.4001	0.4346	0.4065	0.4111	0.3996	0.4226	0.3956	0.4141	0.3924
Falon7b	0.5484	0.4666	0.5826	0.3865	0.6035	0.3496	0.5822	0.3568	0.6024	0.3562	0.6086	0.3570
Falon11b	0.6810	0.5666	0.7044	0.4983	0.7096	0.4756	0.6911	0.4689	0.7039	0.4650	0.6993	0.4647
Falon40b	0.6059	0.5682	0.6508	0.4402	0.6820	0.4492	0.6689	0.4255	0.6870	0.4221	0.6927	0.4277
LLaMA-2-7B	0.6125	0.5091	0.7051	0.4667	0.6990	0.4416	0.6878	0.4378	0.7059	0.4355	0.7067	0.4320
LLaMA-3.1-8B	**0.7074**	0.6038	**0.7856**	0.5598	**0.7885**	0.5005	**0.7733**	0.4970	**0.7879**	0.4850	**0.7853**	0.4768
LLaMA-2-13B	**0.7111**	0.6010	**0.7547**	0.5353	0.7707	0.4896	**0.7521**	0.4870	**0.7715**	0.4815	0.7616	0.4780
LLaMA-3.1-70B	**0.6921**	0.6108	**0.7790**	0.5492	**0.7784**	0.4893	**0.7703**	0.4918	**0.7957**	0.4761	**0.7911**	0.4673
Text-embedding-3-ada-002	0.6163	**0.7439**	0.6125	**0.8011**	0.5726	**0.8479**	0.5755	**0.8255**	0.5628	**0.8496**	0.5610	**0.8421**
Text-embedding-3-small	0.5296	**0.7624**	0.5142	**0.8351**	0.4786	**0.8629**	0.4740	**0.8372**	0.4545	**0.8610**	0.4734	**0.8499**
Text-embedding-3-large	0.6212	**0.7868**	0.6208	**0.8540**	0.5322	**0.8880**	0.5555	**0.8760**	0.5309	**0.8853**	0.5476	**0.8860**
CodeBERT	0.4230	0.4432	0.4771	0.3642	0.4576	0.3572	0.4577	0.3505	0.4499	0.3346	0.4605	0.3281
CodeGPT2	0.5985	0.5463	0.5655	0.5299	0.5538	0.5208	0.5611	0.5186	0.5759	0.5142	0.5641	0.5103
CodeLLaMa-7B	0.6218	0.6410	0.7303	0.5508	0.7747	0.4911	0.7611	0.4990	0.7628	0.4877	**0.7645**	0.4857
CodeLLaMa-13B	0.6565	0.5929	0.7315	0.5397	**0.7842**	0.4817	0.7458	0.4979	0.7635	0.4861	0.7593	0.4821
CodeLLaMa-34B	0.6249	0.5825	0.7255	0.5207	0.7565	0.4925	0.7469	0.4972	0.7552	0.4959	0.7627	0.4832
CodeLLaMa-70B	0.5546	0.5913	0.6869	0.5358	0.7216	0.4588	0.7138	0.4781	0.7226	0.4624	0.7371	0.4568

- Trillion-Scale Models (Text-embedding-3 Series): Similar to billion-scale models, these models show a decreasing trend in PL tasks, though the decline in PT performance is slightly slower than the decline in PL performance. In general, they perform consistently well on PT tasks, as indicated in bold in the table, consistently ranking among the top three models. Furthermore, they demonstrate stable and top-tier performance in PL prediction, suggesting that embedding-based approaches effectively capture both the syntactic and semantic properties of code snippets.

The box plots in Fig. 6 show a higher variance at smaller sizes (5, 10) that decreases as the size increases, plus a consistently lower variance for text-embedding-3 models, reinforcing their stability. SVM curves show higher medians and tighter spreads than KNN across models.

In conclusion, the Text-Embedding-3 models appear to be the most effective models, achieving higher accuracy, lower variance, and stronger performance across both PL and PT tasks. In addition, the general trends and performances across different classifiers for these models remain consistent.

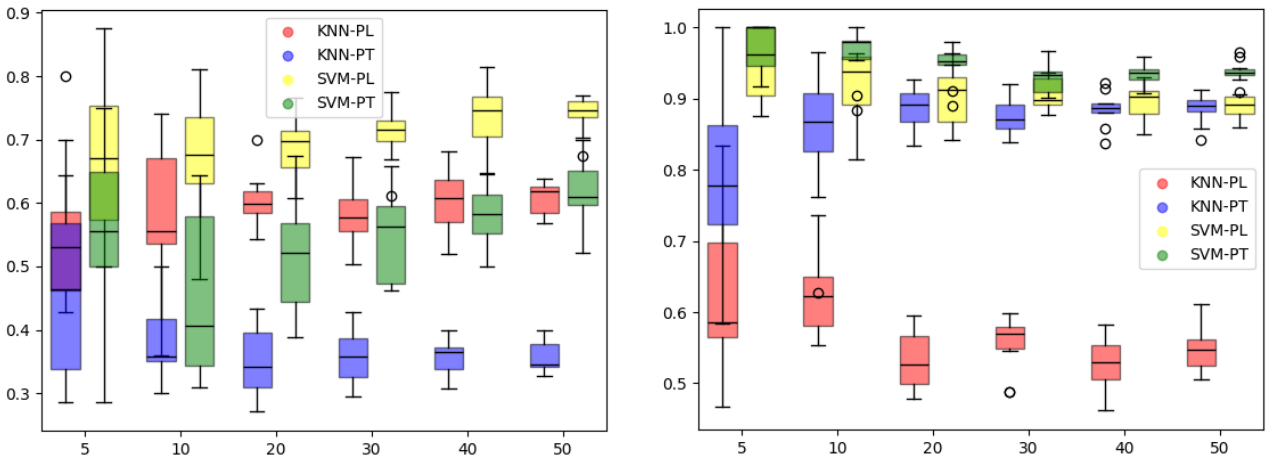
> **RQ1.1:** *Million- and billion-scale language models are more likely to capture linguistic features. In contrast, trillion-scale models group programs by task similarity, providing us with the possibility to recognize the purpose of the code.*

C. Experiment 3

General-purpose models predict the task descriptions using the PT classifier: As shown in the columns 'text' of Table V, the precision of various models is evaluated in three classifiers (KNN, SVM, CNN) for three prediction tasks (PL, PT, and text). Text-embedding-3 models consistently excel in text

TABLE VII: Results of Experiment 2: Median Accuracy from Ten Random Selections across Six Sample Sizes Using SVM. Bold text highlights the top 3 highest accuracy for each classification task.

SVM	5		10		20		30		40		50	
Model	PL	PT	PL	PT	PL	PT	PL	PT	PL	PT	PL	PT
BERT	0.8981	0.8670	0.7960	0.8118	0.7946	0.7612	0.7546	0.7323	0.7566	0.7487	0.7394	0.7423
GPT2	0.5429	0.4491	0.4389	0.2936	0.4249	0.3081	0.4005	0.3147	0.4305	0.3294	0.4304	0.3487
RoBERTa	0.8417	0.7998	0.7776	0.7561	0.7933	0.7324	0.7709	0.7070	0.7840	0.7065	0.7675	0.7095
Falon7b	0.6698	0.5507	0.6669	0.4533	0.6871	0.5175	0.7102	0.5458	0.7361	0.5785	0.7396	0.6185
Falon11b	0.9512	0.8995	0.8940	0.8742	0.8861	0.8540	0.8697	0.8409	0.8696	0.8550	0.8674	0.8498
Falon40b	0.9580	0.8467	0.8860	0.8449	0.8964	0.8326	0.8754	0.8279	0.8782	0.8468	0.8708	0.8403
LLaMA-2-7B	0.9497	0.8916	0.9115	0.8733	0.8845	0.8486	0.8752	0.8500	0.8738	0.8620	0.8677	0.8527
LLaMA-3.1-8B	0.9652	0.9232	**0.9293**	0.8916	**0.9193**	0.8662	0.9025	0.8583	0.8978	0.8764	0.8891	0.8621
LLaMA-2-13B	**0.9789**	0.9279	0.9212	0.8933	0.9127	0.8703	0.8925	0.8601	0.8880	0.8746	0.8844	0.8648
LLaMA-3.1-70B	0.9650	0.9233	0.9278	0.9036	0.9177	0.8606	**0.9120**	0.8576	**0.9074**	0.8781	0.8994	0.8613
Text-embedding-3-ada-002	0.9443	**0.9768**	0.9119	**0.9471**	0.9084	**0.9378**	0.8933	**0.9161**	0.8926	**0.9284**	0.8814	**0.9239**
Text-embedding-3-small	0.9024	**0.9614**	0.8724	**0.9289**	0.8706	**0.9266**	0.8569	**0.9021**	0.8513	**0.9177**	0.8363	**0.9114**
Text-embedding-3-large	0.9515	**0.9768**	0.9211	**0.9629**	0.9037	**0.9478**	0.9062	**0.9290**	0.8959	**0.9349**	0.8893	**0.9387**
CodeBERT	0.8179	0.7284	0.7569	0.6842	0.7879	0.6776	0.7584	0.6562	0.7764	0.6554	0.7687	0.6529
CodeGPT2	0.9057	0.8836	0.8442	0.8504	0.8170	0.8287	0.7979	0.8102	0.8067	0.8118	0.7828	0.8118
CodeLLaMa-7B	**0.9861**	0.9349	0.9217	0.9038	0.9074	0.8713	0.9087	0.8627	0.8997	0.8857	0.8951	0.8754
CodeLLaMa-13B	0.9662	0.9236	**0.9349**	0.9018	**0.9190**	0.8776	**0.9140**	0.8747	0.8999	0.8841	**0.9043**	0.8797
CodeLLaMa-34B	**0.9784**	0.9353	**0.9349**	0.8998	**0.9196**	0.8608	**0.9135**	0.8732	**0.9048**	0.8882	**0.9059**	0.8800
CodeLLaMa-70B	0.9668	0.9245	0.9187	0.8835	0.9119	0.8632	0.9099	0.8648	**0.9012**	0.8793	**0.9047**	0.8723



(a) Falcon-7b (b) Text-embedding-3-large

Fig. 6: Accuracy from Ten Random Selections of Languages and Tasks for Falcon-7B and Text-embedding-3-large Embeddings

classification across all classifiers (bold numbers in the text columns of the table), highlighting their strong ability to understand both code and natural language representations.

Among the classifiers, SVM consistently outperforms KNN and CNN in all three tasks (e.g., LLaMA-3.1-8B achieves 0.5338 accuracy in text classification with SVM, a significant improvement over BERT's 0.1192). This demonstrates SVM's effectiveness in feature separation for natural language recognition. Notably, SVM achieves the highest accuracy in text classification, further reinforcing its strength in handling textual data. While million-scale models show only minor improvements under SVM, billion-scale models benefit significantly, particularly in text classification, narrowing the performance gap among PL, PT, and text tasks.

Conversely, CNN performs the worst, particularly in PT and text classification. While its PL accuracy is reasonable for some models, it lags behind SVM. The poor text classification accuracy of CNN suggests that our customized architecture is not sufficiently advanced for the multidimensional embeddings.

Across all classifiers, a clear performance distinction is observed among models of million-scale, billion-scale, and trillion-scale sizes. As model size increases, text prediction performance improves significantly. For example, LLaMA-3.1-8B achieves 0.5338 in text classification using SVM, a substantial improvement over BERT's 0.1192. Text-embedding-3 models consistently perform well across all tasks, reinforcing their superior ability to capture both code and natural language semantics.

RQ1.2: *Smaller LLMs encode programming tasks more reliably from code than from descriptions, revealing a gap in how they represent meaning across the two modalities. In contrast, the text-embedding-3 models encode code and descriptions in similar ways within the latent space, creating a strong connection between the two modalities and demonstrating better semantic alignment.*

RQ1: *Overall, only trillion-scale models can fully achieve literate programming. Factors affecting performance may include, but are not limited to, the size of models and the maximum input length.*

D. Experiment 4

Language model trained on code: As shown in Table V. In contrast, code-specific models generally perform equally well with general-purpose LLMs of the same size; their improvements are more promising in predicting PT, whereas text prediction accuracy shows a slight decline. Comparing the best-performing code-specific models (bold number in the table) with general-purpose models (Table V) at the same scale, we only observe improvements in predicting PL (around 1% - 3%) but do not observe significant improvements in text prediction, suggesting that domain-specific training does not necessarily enhance text interpretation outside of code-related contexts. Furthermore, when comparing code-specific models to the top-performing general-purpose models, no clear improvement is evident, as Text-embedding-3 series models outperform all other models. For PLMs, CodeBERT and CodeGPT2 exhibit mixed performance. CodeBERT performs relatively poorly compared to modern models, while CodeGPT2 shows noticeable improvement, particularly in PT accuracy. This suggests that the effectiveness of code-specific architectures depends on their training and fine-tuning at this scale.

From Table VI and Table VII, we observe that code-specific models help narrow the gap between PL and PT, reducing variance across tasks. For example, LLaMA-3.1-8B achieved 0.8552 accuracy on PL versus 0.7410 on PT with SVM, a gap of 0.11, while CodeLLaMA-34B scored 0.8664 (PL) versus 0.7634 (PT), a gap of 0.10. This suggests that specialized training enhances consistency in programming-related predictions, but not for text prediction. Regarding the CodeLLaMA series, these models perform exceptionally well in predicting PL. CodeLLaMa-13B/34B/70B rank among the top three when using SVM classifiers, compared to general-purpose models. However, box plots reveal that improvements in other tasks remain minor.

When comparing box plots of different models at the same sizes, despite fluctuations in values, we do not observe distinct patterns or trends for CodeLLaMA models. For BERT and CodeBERT, CodeBERT introduces greater variation in the plots, while BERT reduces the differences between PL and PT as the sample size increases.

RQ2: *Compared to general-purpose models, code-specific models improved the accuracy of predicting programming languages, tasks, and program descriptions; however, the performance gaps between these tasks remain substantial. As model size increases, the Text-embedding-3 series models consistently outperform all code-specific models, achieving significant gains across all tasks.*

E. Validation: CodeNet

We validate our findings on the CodeNet dataset, which contains 55 programming tasks and 52 programming languages, but nearly 70 times more records than Rosetta Code. Due to computational limitations, we restricted the parameter size for KNN and were unable to train SVM classifiers on the full dataset. The results have been summarized in Table VIII.

The overall patterns observed on CodeNet are broadly consistent with those on Rosetta Code. First, larger or more recent models such as the LLaMA and Falcon families consistently outperform earlier architectures like BERT, GPT-2, and RoBERTa, reflecting the benefits of both scale and training methodology. Second, the Text-embedding-3 series remains dominant for tasks involving textual descriptions, confirming that embedding-optimized objectives are more effective for capturing semantic alignment across modalities. Finally, as with Rosetta Code, programming language and programming task classification are generally easier than textual description prediction, underscoring the challenge of aligning natural language with code semantics.

However, significant differences emerge when comparing the validation results on CodeNet with the evaluation results on Rosetta Code. These differences span across classifiers, where the best-performing method shifts between datasets; dataset characteristics, with accuracy generally higher on the larger-scale CodeNet; and model types, where general-purpose and code-specific LLMs exhibit distinct strengths.

With KNN classifiers, PL classification remains relatively high across models, with billion-parameter LLMs approaching similar accuracy regardless of architecture. This confirms that larger models encode strong signals of programming language identity. In contrast, PT classification is weaker and more variable, reflecting the greater difficulty of capturing task semantics.

A notable difference between datasets is the role of the classifier. On Rosetta Code, SVM consistently yielded the best performance, whereas on CodeNet, CNN achieved the highest accuracy. This suggests that classifier effectiveness depends strongly on dataset scale and distribution, rather than being universally optimal. Accuracy levels were also much

TABLE VIII: Validation of Experiment 1, 3, and 4: PL and PT Classification Accuracies Using Embeddings. Bold text highlights the top 3 highest accuracy for each classification task.

	Classifier	KNN			CNN		
	Accuracy	PL	PT	text	PL	PT	text
General-purpose Models	BERT	0.8520	0.7270	0.3636	0.9134	0.8421	0.1636
	GPT2	0.6261	0.3931	0.0364	0.9386	0.7473	0.1818
	RoBERTa	0.7982	0.6420	0.0364	0.9095	0.7368	0.0000
	Falon7b	0.9084	0.6096	0.0545	0.9490	0.8213	0.4364
	Falon11b	0.9211	0.6458	0.1273	0.9441	0.8081	0.3273
	Falon40b	0.9156	0.6310	0.0545	0.9457	0.7253	0.2727
	LLaMA-2-7B	0.9221	0.6519	0.0909	0.9485	0.9123	0.3636
	LLaMA-3.1-8B	**0.9375**	0.6721	0.2000	**0.9534**	0.9260	0.6364
	LLaMA-2-13B	0.9298	0.6601	0.0545	0.9507	0.9309	0.6182
	LLaMA-3.1-70B	0.9178	0.6228	0.1273	0.9386	0.8893	0.6000
	Text-embedding-3-ada-002	0.8972	**0.8384**	**0.9818**	0.9379	0.9274	**0.8182**
	Text-embedding-3-small	0.8980	**0.7412**	**0.9636**	0.9391	0.9134	**0.7818**
	Text-embedding-3-large	0.8975	**0.8492**	**0.9818**	0.9430	**0.9485**	**0.9273**
Models for Code	CodeBERT	0.7971	0.5976	0.0727	0.8810	0.6968	0.2000
	CodeGPT2	0.8843	0.6404	0.2000	0.9441	0.8575	0.2909
	CodeLLaMa-7B	**0.9359**	0.6617	0.0364	0.9518	**0.9331**	0.5455
	CodeLLaMa-13B	0.9315	0.6557	0.0364	0.9518	0.9326	0.6000
	CodeLLaMa-34B	**0.9331**	0.6393	0.0727	**0.9539**	**0.9496**	0.7455
	CodeLLaMa-70B	0.9254	0.6299	0.0364	**0.9561**	0.9315	0.5818

higher in validation than in evaluation, showing that larger datasets with more examples per class help models expose their representational capacity more effectively.

Another key trend concerns model types. General-purpose LLMs struggled heavily with text description classification on Rosetta Code (often near 0), yet performed much better on CodeNet, where training data richness may have allowed embeddings to generalize. Code-specific models, especially the CodeLLaMA series, scaled more effectively in validation, achieving much higher CNN performance than on evaluation. By contrast, the Text-embedding-3 series models maintained dominance in PT and textual classification across both datasets, consistently ranking among the top three models. Their embedding-optimized training objective appears to provide robust semantic alignment, a property central to our definition of literate programming.

Taken together, these results reinforce three conclusions: (i) PL classification is consistently easier than PT classification, supporting RQ1.1; (ii) embedding-optimized models (Text-embedding-3) outperform both general-purpose and code-specific LLMs in task-related evaluations, supporting RQ1.2; and (iii) code-specific models improve when scaled and validated on larger datasets, but do not surpass embedding-optimized models in semantic alignment, consistent with RQ2.

We also evaluate Experiment 2 using the validation dataset. Based on the ratio of randomly chosen programming languages and programming tasks, instead of 6 different sizes, we only select two sizes: 5 and 10 for random choice.

We validated Experiment 2 (Table IX) on the CodeNet dataset using reduced random sample sizes (5 and 10). The reduced random sample is due to the smaller size of programming languages and programming tasks in the validation dataset. Compared to Rosetta Code, overall accuracies are substantially higher, reflecting the benefit of CodeNet's larger scale and richer data distribution. For instance, even baseline models such as BERT and RoBERTa achieved PT classification above 0.70 with SVM, far surpassing their evaluation results.

Classifier effectiveness also differed between datasets. On Rosetta Code, SVM was consistently the strongest probing method, while KNN often lagged. On CodeNet, however, both SVM and KNN performed competitively, with SVM showing robust results across nearly all models. This suggests that the larger, more balanced validation dataset enabled classifiers to utilize embeddings more effectively, thereby reducing overfitting.

Taken together, these results show that: (i) validation accuracy is consistently higher than evaluation, underscoring the importance of dataset scale; (ii) classifier effectiveness depends on dataset characteristics, with SVM clearly excelling on CodeNet; (iii) embedding-optimized models remain superior for task semantics, while code-specific and LLaMA models scale effectively for language classification. These findings reinforce RQ1.1 and RQ1.2 while further supporting RQ2, showing that code-specific models become competitive at scale but still do not surpass embedding-optimized models in task-level recognition.

> **Validation** *on the CodeNet dataset, which is substantially larger but less diverse than Rosetta Code, confirmed our main findings: syntactic features are easier to capture than semantic ones by smaller models, with text-embedding-3 models best at encoding both information in latent space.*

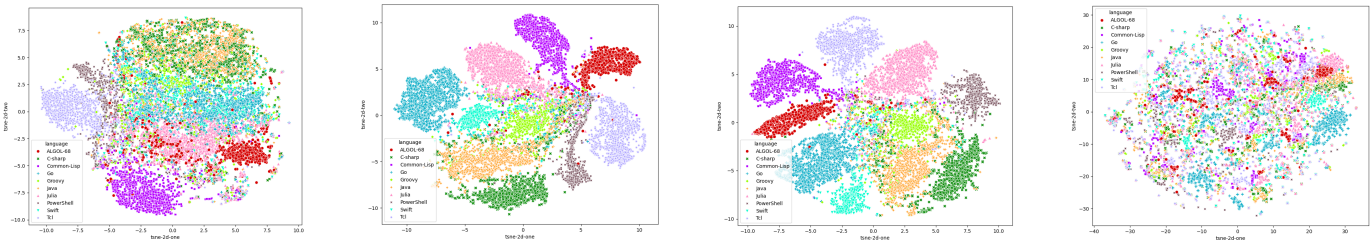
F. Visualization

To enhance visualization, we sampled data from 10 programming languages, ranging from low-level to high-level abstraction languages such as Java. The resulting visualization is shown in Fig. 7 (Plots for all models are included in the

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3629828

TABLE IX: Validation of Experiment 2: Median Accuracy from Ten Random Selections across Two Sample Sizes Using KNN and SVM. Bold text highlights the top 3 highest accuracy for each classification task.

		KNN				SVM			
		5		10		5		10	
	Model	PL	PT	PL	PT	PL	PT	PL	PT
General-purpose Models	BERT	0.8899	0.8566	0.9103	**0.8020**	**0.9895**	0.9703	0.9747	0.9287
	GPT2	0.6619	0.5552	0.6743	0.4616	0.9070	0.7849	0.9396	0.8490
	RoBERTa	0.7891	0.7490	0.8374	0.7136	0.9770	0.9411	0.9627	0.8926
	Falon7b	0.9262	0.7774	0.9576	0.6820	0.9622	0.8699	0.9607	0.9035
	Falon11b	0.9569	0.8026	0.9630	0.7024	**0.9947**	0.9632	0.9707	0.9630
	Falon40b	0.9445	0.7978	0.9711	0.7074	**1.0000**	0.9457	0.9802	0.9416
	LLaMA-2-7B	0.9622	0.7807	0.9709	0.7035	**0.9895**	0.9603	0.9873	0.9647
	LLaMA-3.1-8B	**0.9828**	0.8039	**0.9771**	0.7330	**0.9895**	0.9730	**0.9902**	**0.9776**
	LLaMA-2-13B	**0.9828**	0.7877	0.9710	0.7148	**0.9895**	0.9653	0.9873	0.9654
	LLaMA-3.1-70B	**0.9705**	0.7209	**0.9799**	0.7003	0.9788	0.9299	**0.9938**	0.9340
	Text-embedding-3-ada-002	0.5909	**0.9436**	0.3739	**0.9031**	0.5970	**1.0000**	0.3806	**0.9909**
	Text-embedding-3-small	0.9141	**0.8847**	0.9453	0.7840	0.9659	**0.9913**	0.9889	0.9710
	Text-embedding-3-large	0.9202	**0.9690**	0.9435	**0.8953**	0.9659	**1.0000**	**0.9902**	**0.9897**
Models for Code	CodeBERT	0.8333	0.7610	0.8665	0.6918	0.9694	0.9302	0.9730	0.8733
	CodeGPT2	0.9195	0.7768	0.9255	0.6856	0.9746	0.9563	0.9812	0.9340
	CodeLLaMa-7B	0.9604	0.7632	**0.9735**	0.6866	0.9830	0.9840	0.9855	0.9659
	CodeLLaMa-13B	0.9644	0.7546	0.9722	0.6719	0.9788	0.9657	0.9839	0.9689
	CodeLLaMa-34B	0.9562	0.7602	0.9721	0.6534	0.9788	0.9698	0.9790	0.9585
	CodeLLaMa-70B	0.9551	0.7978	0.9695	0.6540	0.9832	0.9655	0.9809	0.9591



(a) BERT (b) LLaMA-2-7B (c) CodeLLaMa-7B (d) Text-embedding-3-large

Fig. 7: t-SNE Representations of Embeddings in Two Dimensions

Supplementary Data). Most languages form distinct clusters, and as model size increases, these clusters become more pronounced.

However, Text-embedding-3 models do not exhibit apparent clustering. This may be due to the choice of perplexity, as perplexity heavily influences the formation of clusters in t-SNE plots. Additionally, reducing high-dimensional embeddings to just two dimensions may obscure strong clustering patterns. Furthermore, the vector representation of different programming tasks remains unclear. Therefore, while these plots help us understand the embeddings, they are not used as conclusive evidence.

VI. Discussion

a) Dataset: We selected two large datasets with many programming languages and programming tasks. That choice allowed us to test the hypothesis and establish that only the largest language models are capable of literate programming. This means that programmers can, with a high degree of certainty, rely on the generated programs solving the right tasks in the right programming language. The majority of the existing benchmarks are too small in terms of either too few programming languages (only a handful) or too few programming tasks. This does not mean that the benchmarks are wrong, they are just too small for our purposes.

Regarding input data for classifiers (the output embeddings of chosen models), which consist of code and programming text embeddings, their length depends on the model architecture, ranging from 768 to 8,192 dimensions. This variation in embedding size may affect the accuracy. However, the Text-embedding-3-large model, with an embedding size of 3072 (a medium-sized representation), consistently achieved stable and high accuracy across all classification tasks. This suggests that the selected classifiers can handle high-dimensional inputs and that embedding size alone is not the primary cause of inefficiencies. Consequently, KNN emerges as the most cost-effective classifier for our study.

b) Classifier: We intentionally chose lightweight probing classifiers (SVM, KNN, CNN) for interpretability. More complex classifiers (e.g., transformers on embeddings) could yield higher performance, but would obscure whether embeddings (and thus the models) themselves contain the required information. The classifiers differ in performance, but they show a similar trend, which means that the results from the classifications is based on the information in the embeddings rather than based on the classifiers' ability to find similarities between vectors.

c) Text-Embedding-3 Series Models: Because OpenAI has not disclosed detailed architectural information, our analysis in Section V is limited to known specifications and observed performance. Another possible explanation for the strong results of the text-embedding-3 series lies in their likely training objectives. Embedding models are generally optimized to produce vector representations that capture semantic similarity between inputs, which makes them particularly effective for tasks such as clustering, classification, and retrieval. In contrast, general-purpose LLMs (e.g., GPT, LLaMA, Falcon) are primarily trained for next-token prediction, an objective that emphasizes fluency in text generation rather than enforcing semantic alignment in the embedding space. Embedding-oriented models are often trained with contrastive or retrieval-based objectives that encourage semantically similar inputs to be placed closer together in latent space. While this may explain the superior performance we observe, confirming this would require further study, ideally with transparent training details or controlled experiments.

d) Programming Language Recognition: PL classification was consistently easier for models than PT classification. All models performed well in this task ($ACC_{PL} \geq ACC_{PT}$ -0.02 when the classifier was SVM). This trend reflects that *syntactic regularities and lexical markers* languages are strong signals, which larger models are particularly adept at capturing, and this aligns with the naturalness hypothesis proposed by Hindle et al. [11]. For billion-scale models, the accuracy improved as the model size increased, except for an anomaly where LLaMA-3.1-8B outperformed LLaMA-3.1-13B. The anomaly where LLaMA-3.1-8B outperformed LLaMA-3.1-13B can be attributed to training efficiency and data quality: the smaller and newer models benefit from *more refined training corpora and architectural improvements*, sometimes exceeding the larger ones. Thus, while size generally improves PL recognition, training methodology matters as much as parameter count. Therefore, if we want to understand code alone, LLaMA-3.1-8B would be the optimum choice if computational resources are limited.

This explains why models achieve near-perfect PL recognition but still fall short in PT classification: language recognition exploits surface-level syntax, whereas task recognition requires capturing deeper semantic intent. These contrasting results directly address RQ1.1 by showing that while LLMs encode syntactic regularities effectively, their ability to represent task-level semantics remains limited.

e) Programming Tasks Recognition: PT classification was significantly more complicated based on the embeddings. Even trillion-scale general-purpose models struggled to match the performance of models optimized for embedding. The Text-embedding-3 series dominated task classification. This advantage arises from their training objective, which is explicitly optimized for semantic similarity in embedding space, unlike autoregressive LLMs trained for next-token prediction. By encoding semantically similar programs and descriptions closer together, Text-embedding-3 models exhibit the kind of semantic alignment that is central to literate programming.

This finding also clarifies why code-specific models (e.g., CodeBERT, CodeLLaMA) lag in PT recognition: while fine-tuning on source code strengthens structural and syntactic representations, it does not necessarily improve the abstract mapping between problems and solutions.

f) Textual Description Recognition: Textual description prediction further reinforced this distinction. The Text-embedding-3 models not only achieved the highest accuracy, but also showed much smaller performance gaps between PL, PT, and text. This indicates that they maintain consistent representations across modalities, capturing both natural and programming languages in a unified space. In contrast, CNN classifiers failed in text descriptions, suggesting that convolutional architectures do not align well with embedding structures optimized for semantic similarity.

These results directly address RQ1.2: LLMs do not represent tasks and descriptions equally well unless explicitly optimized for embeddings. General-purpose and code-specific models tend to prioritize code syntax, while embedding-optimized models capture task-level semantics more effectively.

g) General vs. Code-Specific Models: Our results also address RQ2. Code-specific models consistently perform well in PL classification, sometimes rivaling general-purpose billion-scale models, confirming that they effectively capture syntactic and structural properties of source code. However, they show no clear advantage in PT or textual tasks, and their gap with general-purpose LLMs decreases as the scale increases. *This suggests that code-specific training enhances syntactic recognition, but does not substantially improve semantic alignment*—which is critical for literate programming. In contrast, large general-purpose LLMs, especially those optimized for embedding, encode language and task semantics more effectively.

Implications for Software Engineering Practice: Our findings carry several implications for the design and use of LLM-based tools in software engineering:

1) Code completion versus task recognition. Since most LLMs excel at recognizing programming languages (syntax), current tools such as Copilot and IntelliCode benefit from strong syntactic fluency. However, their limited ability to capture programming tasks explains why generated code often lacks semantic correctness or problem alignment. It also shows why we need to complement LLMs with agentic AI and RAGs when designing these tools.

2) Code Summarization and Documentation. Embedding-optimized models (e.g., Text-embedding-3) demonstrate superior alignment between code and textual descriptions. This suggests that they are better suited for tasks like automated documentation, summarization, and explanation, the core aspects of literate programming. Integrating these embeddings into SE tools could improve explainability and maintainability.

3) Cross-Language and Multi-Task Scenarios. Because task recognition is harder than language recognition, tools for cross-language code search, translation, or multi-language

repositories should prioritize models that explicitly encode semantic similarity (e.g., embedding-based models).

4) Choice of Models under Resource Constraints. Our results show that smaller, well-trained models (e.g., LLaMA-3.1-8B) can rival or exceed older, larger models, making them practical choices for organizations with limited compute. However, for task-level semantics, embedding-optimized models remain the preferred option when resources permit.

In short, software engineering practitioners should match the choice of model with the intended task: code-specific or syntactic tools for completion and bug detection, and embedding-optimized models for summarization, documentation, or semantic code search.

VII. Validity evaluation

We use the framework by Wohlin [59] to evaluate the threats to the validity of our experiments. In this computational experiment, we focused on maximizing the conclusion and external validity.

In our case, we focused on obtaining results that are as generalizable as possible, thus prioritizing the **external validity**. Using programs from Rosetta Code, we minimize the risk of studying a non-representative sample of programs. Rosetta Code is a community-driven website that collects implementations of the same problems done by skilled programmers in that particular language. This means that the implementations adhere to the best practices for each programming language. The repository contains multiple solutions for the same problem in the same programming language, if there are numerous. Although it is only around 40,000 programs, it is diverse enough to reduce the risk that our results are biased toward a specific programming style, language, or problem.

To achieve as high **conclusion validity** as possible, we used statistics for the evaluation of the hypotheses. We highlighted the best three results for each task instead of one to enhance the degree of significance of our observed results. One threat to the validity of conclusions is related to the ability of classifiers to learn from high-dimensional embeddings. If simple classifiers fail to generalize, our conclusions about the quality of embeddings may be compromised. Conversely, if we use highly complex models, it becomes unclear whether performance improvements stem from the embeddings or the classifier itself. This presents a trade-off between classifier complexity and effective evaluation. To strengthen our conclusions, we incorporate multiple classifiers in our experiments, ensuring that our findings are not biased toward a specific model choice. We also include KNN, a simple classifier with minimal hyperparameters, and analyze trends rather than relying solely on accuracy.

However, a potential threat to **construct validity** arises from our decision to operationalize literate programming through probing classifiers applied to embeddings of existing programming repositories. Although this approach avoids confounding factors such as prompt leakage and variability in prompt engineering, it evaluates 'literacy' indirectly through classification accuracy rather than through end-to-end code generation or explanation. Consequently, high probing accuracy may indicate that embeddings capture discriminative features of programming languages and tasks, but not necessarily that models can generate coherent, literate explanations in practice. Thus, our measure of literate programming alignment captures an important but partial view of the construct, and results should be interpreted with this limitation in mind.

Additionally, we had to choose the LLMs in our study so that they are comparable to each other (in terms of size of the embedding space) and can be used for both NLP tasks and programming tasks. This means that we studied models with ca. 100 million parameters (compared to an estimated more than 7 billion parameters for LLaMA models or a trillion Text-embedding-3-large). Although this is a constraint, using large models would introduce a confounding factor – models being biased toward specific programming languages or problems (since pretraining on 40,000 programs is ineffective for such large models). Another tradeoff in our study is to use embeddings rather than instructing the models to generate programs. This is a direct consequence of using statistical models in large corpora of programs; however, it means that we cannot determine whether the models can still accurately generate code for a given problem in a specific programming language.

Internal validity in our study is naturally a trade-off. To test the hypothesis, we need to assume that it is equally easy to identify a solved problem as it is to identify the programming language. This seems like an easy assumption, but it does not have to be. For example, if we take Java as the programming language and Fibonacci as the problem, most modern programmers would be able to recognize both. However, if we take an 8086 assembly and a program that copies strings, programmers may have a problem recognizing both (e.g., which assembly it is or whether it is copying the string or searching in a string). This means that there is no guarantee of an interaction between the cause and the effect – it may not be observable by humans. So, since they are "observable" by the computer (after compilation/interpretation, the program works correctly), some of the languages may have an inherent bias towards the programming language. There is no taxonomy of programming languages (or even order) regarding how close they are to human understanding, and therefore to the original programming literacy of Knuth. Thus, to minimize this threat, we conducted two experiments with two models with high-level languages only. We also conducted experiments with models that were not trained on programs at all to avoid bias towards the programming languages. Experiments that can compare human performance in recognizing PT and PL are planned for our future work.

VIII. Conclusions

Literate programming, originating in the work of Knuth [1], says that software programs should be written in a way understandable to humans and machines alike. Over time, this concept has been implemented in various ways, with

one widespread practice being the Jupyter Notebook, where figures, Markdown syntax, comments, and code coexist. Modern LLMs seem to be able to provide both explanations of solutions to programming problems in a similar way when prompted; the explanation and the code coexist in the same answer. However, it is not clear whether these explanations are equally suitable for both the source code and the task.

In this study, we conducted a year-long investigation into whether LLMs are capable of literate programming, that is, whether they can explain both the programming language and the task of a given program equally well. To test this hypothesis, we designed four experiments using 19 transformer-based models on the Rosetta Code repository, covering 1,228 tasks across 926 programming languages and more than 3,300 trials. We further validated our findings on the larger CodeNet dataset, which comprises 55 tasks and 52 languages, with nearly 70 times more records, resulting in over 2,000 additional trials. Although the CodeNet experiments were substantially more time- and resource-intensive, the Rosetta Code repository offered greater diversity, spanning a wider range of programming tasks and languages.

Our results indicate that trillion-scale models are the only ones that come close to literate programming by consistently recognizing both the programming language and the programs' corresponding tasks with an accuracy of over 70% on both datasets. These models perform well even with limited data and outperform smaller, but code-specific models. For smaller LLMs, the ability to recognize programming languages and tasks improves with model size. These smaller LLMs perform acceptably only when the set of programming languages and tasks is limited to a handful of languages and tasks. As the variety of tasks and languages increases, performance declines, with task recognition accuracy dropping more sharply than programming language recognition. These findings mean that we can rely on the really large language models, and the small models should be used with caution.

IX. Limitation and Future Work

Limitation: One key limitation of this study is the use of dataset. We restricted experiments to Rosetta Code and CodeNet. While they cover diverse tasks and languages, they do not represent all programming paradigms (e.g., large-scale frameworks, domain-specific languages). Results may not generalize fully to industrial repositories because the industrial repositories contain significantly larger code bases, spread over multiple files, and solve multiple tasks in one system. Studying such repositories is a part of our further work.

The second limitation is the computational constraints. Due to resource limitations, we were unable to train all classifiers (e.g., SVM) on full CodeNet data and restricted KNN parameters[8]. This may have limited absolute performance, although relative trends were consistent between datasets.

[8]We had access to CPUs with 256GB of memory, which was too little for that classifier and the extensive CodeNet data

The third limitation arises from the undisclosed architecture and details of the Text-embedding-3 series models from OpenAI. Since crucial information such as model size, architecture, and training data remain unknown to the public, we are unable to fully investigate the factors that contribute to performance differences beyond the observable ones. This prevents us from making a comprehensive analysis of how specific architectural choices or training methodologies impact model effectiveness, limiting our ability to gain deeper insights into their strengths and weaknesses.

A last limitation is that we evaluate LLMs using embeddings rather than directly analyzing code and text generated from prompts. Our choice was deliberate: generation-based evaluation introduces several biases, including prompt leakage, reliance on surface-level similarity metrics, and susceptibility to prompt-engineering effects that are difficult to control. By contrast, probing embeddings from existing programming datasets allows us to rely on explicit PL and PT labels, providing a controlled and reproducible evaluation. Nevertheless, if these biases can be mitigated, future work should complement our approach with generation-based studies, directly comparing code and description outputs to assess literate programming capabilities.

Future Work: Several directions arise naturally from the limitations of this study:

A first direction for future work is to explore generated code and descriptions through prompt. Rather than relying solely on pre-existing datasets, future studies could leverage Agentic AI to generate both code and corresponding textual descriptions, while carefully mitigating the influence of prompt design. This would allow for a deeper analysis of whether models can generate both code and descriptions with consistent semantics, providing insight into their alignment between code recognition and natural language generation.

Second direction is expanding datasets. Future research should extend beyond Rosetta Code and CodeNet to include large-scale industrial repositories, domain-specific languages, and multi-module frameworks. This would test whether the literate programming hypothesis for LLMs generalizes to real-world software engineering environments where tasks are larger and more context-dependent.

Thirdly, we can bridging probing and generation. future studies should move from static probing of embeddings to evaluating LLMs in end-to-end generation of code and explanations. Developing reliable automatic evaluation pipelines—for example, combining executability checks with semantic equivalence metrics—would allow researchers to test whether the internal literate programming alignment we observe translates into practical improvements in generated code and documentation.

Acknowledgements

Sincere gratitude is expressed to Prof. Premkumar Devanbu and Dr. Toufique Ahmed for their invaluable assistance in shaping the research questions and designing the experiments.

This project required more than 10,000 GPU hours, and we extend our deep appreciation to Prof. Christian Berger, Dr. Srijita Basu, and Pierre Lamart for generously providing access to their GPU resources, without which these experiments would not have been possible.

Finally, we thank the anonymous reviewers for their thoughtful comments and valuable discussions, particularly on the definition of literate programming, which helped us refine the research focus of this paper in the context of LLMs.

The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no.2024-22-1517, no.2025-22-1040, no.2025-22-1115, and no.2025–22-467.

This work has also been partially funded by Software Center, a collaboration between the University of Gothenburg, Chalmers, and 18 universities and companies – www.software-center.se.

The work has been partially funded by the Swedish Research Council under the project 2024-04687.

REFERENCES

[1] D. E. Knuth, "Literate programming," *The computer journal*, vol. 27, no. 2, pp. 97–111, 1984.

[2] K. Shi, D. Altınbüken, S. Anand, M. Christodorescu, K. Grünwedel, A. Koenings, S. Naidu, A. Pathak, M. Rasi, F. Ribeiro *et al.*, "Natural language outlines for code: Literate programming in the llm era," *arXiv preprint arXiv:2408.04820*, 2024.

[3] Y. Belinkov, "Probing classifiers: Promises, shortcomings, and advances," *Transactions of the Association for Computational Linguistics (TACL)*, 2022.

[4] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1332–1336.

[5] J. Bentley and D. Knuth, "Programming pearls: literate programming," *Communications of the ACM*, vol. 29, no. 5, pp. 384–369, 1986.

[6] E. Schulte, D. Davison, T. Dye, and C. Dominik, "A multi-language computing environment for literate programming and reproducible research," *Journal of Statistical Software*, vol. 46, pp. 1–24, 2012.

[7] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook: Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI conference on human factors in computing systems*, 2018, pp. 1–11.

[8] R. Rädle, M. Nouwens, K. Antonsen, J. R. Eagan, and C. N. Klokmose, "Codestrates: Literate computing with webstrates," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017, pp. 715–725.

[9] R. W. Brennan and B. McDermott, "Coding literacy in the age of generative ai," in *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Springer, 2023, pp. 445–456.

[10] 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)*. IEEE, 2023, pp. 31–53.

[11] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[12] R. Gupta and S. Riezler, "Lightly-supervised models for extracting information from historical newspaper advertisements," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[13] A. Köhn, "What's in an embedding? analyzing word embeddings through multilingual evaluation," in *Proceedings of the 20th Nordic Conference on Computational Linguistics (NoDaLiDa)*, 2015.

[14] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni, "What you can cram into a single vector: Probing sentence embeddings for linguistic properties," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018.

[15] J. Hewitt and C. D. Manning, "A structural probe for finding syntax in word representations," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[16] S. Troshin and N. Chirkova, "Probing pretrained models of source code," *arXiv preprint arXiv:2202.08975*, 2022.

[17] T. Ahmed, D. Yu, C. Huang, C. Wang, P. Devanbu, and K. Sagae, "Towards understanding what code language models learned," *arXiv preprint arXiv:2306.11943*, 2023.

[18] K. Chen, Y. Yang, B. Chen, J. A. H. 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)*. IEEE, 2023, pp. 162–172.

[19] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," 2019.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

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

[22] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.

[23] J. Sarzynska-Wawer, A. Wawer, A. Pawlak, J. Szymanowska, I. Stefaniak, M. Jarkiewicz, and L. Okruszek, "Detecting formal thought disorder by deep contextualized word representations," *Psychiatry Research*, vol. 304, p. 114135, 2021.

[24] 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," *arXiv preprint arXiv:2009.08366*, 2020.

[25] G. Little and R. C. Miller, "Translating keyword commands into executable code," in *Proceedings of the 19th annual ACM symposium on User interface software and technology*, 2006, pp. 135–144.

[26] S. Gulwani and M. Marron, "Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 803–814.

[27] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, "Program synthesis from natural language using recurrent neural networks," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03*, vol. 1, 2017.

[28] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," *arXiv preprint arXiv:1709.00103*, 2017.

[29] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[30] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[31] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[32] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[33] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 253–264.

[34] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair. corr abs/1711.07163 (2017)," *arXiv preprint arXiv:1711.07163*, 2017.

[35] Y. Liang and K. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[36] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[37] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[38] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[39] K. Aggarwal, M. Salameh, and A. Hindle, "Using machine translation for converting python 2 to python 3 code," PeerJ PrePrints, Tech. Rep., 2015.

[40] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens *et al.*, "Moses: Open source toolkit for statistical machine translation," in *Proceedings of the 45th annual meeting of the association for computational linguistics companion volume proceedings of the demo and poster sessions*. Association for Computational Linguistics, 2007, pp. 177–180.

[41] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in neural information processing systems*, vol. 31, 2018.

[42] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[43] 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.

[44] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "Codescore: Evaluating code generation by learning code execution," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.

[45] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "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*, 2023, pp. 5673–5684.

[46] 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.

[47] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[48] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," *arXiv preprint arXiv:2302.07867*, 2023.

[49] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[50] "Rosettacode," http://rosettacode.org/, 2014, accessed: 2025-08-13.

[51] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[52] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[53] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[54] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, E. Goffinet, D. Heslow, J. Launay, Q. Malartic, B. Noune, B. Pannier, and G. Penedo, "Falcon-40B: an open large language model with state-of-the-art performance," *URL https://falconllm.tii.ae*, 2023.

[55] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[56] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[57] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[58] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

[59] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

Simin Sun earned her master's degree in Computer Science and Engineering from the Eindhoven University of Technology, the Netherlands, in 2023. She is pursuing a Ph.D. in Software Engineering at Chalmers and University of Gothenburg, Sweden. With over five years of industry experience in software engineering within prominent companies, her research focuses on machine learning for software engineering. Specifically, she utilizes action research to explore and develop new methods and tools grounded in large language models.

Miroslaw Staron is a Professor of software engineering at the Department of Computer Science and Engineering, Chalmers, and University of Gothenburg, Sweden. Prof. Staron is the Editor-in-Chief of Information and Software Technology and has a regular column in IEEE Software (Practitioners Digest). He is also on the editorial boards of IEEE Software and Elsevier Journal of Systems Architecture. Prof. Staron has been active in national bodies such as AI Sweden, AI Competence for Sweden, and Swedsoft. His work has also been recognized as an Excellent Teacher at the University of Gothenburg. His research focuses on software design, metrics, machine learning, and software quality. Prof. Staron authored five books and over 250 peer-reviewed articles in software engineering.