Empirical Computation: Prompting versus Programming

Eric Tang*
CMU, USA

Jing Liu*
MPI-SP, Germany

Marcel Böhme
MPI-SP, Germany

Abstract

Large Language Model (LLM) agents can solve *any* computational problem *without* an algorithm in a runtime *independent* of the computational complexity of that problem. Instead of specifying precisely how to solve problem instance using *programming*, we ask an LLM to solve the problem instance using *prompting*. Outputs are sampled from a distribution rather than generated procedurally.

In this vision paper, we explore the challenges and opportunities of this new form of computation and observe that its capabilities and limits *cannot* be understood within the classic, rationalist framework of computation. Hence, we appeal to the software engineering (SE) community to develop the foundations and techniques required to analyze the properties of this "empirical computation" as it generates solutions to computational problems: How can we analyze and improve the correctness of LLMs solving a computational problem in the general, in the problem-specific, or in the instance-specific? What are the properties and fundamental limits of empirical computation? This paper aims to establish empirical computation as a field in SE that is timely and rich with interesting problems.

ACM Reference Format:
Eric Tang, Jing Liu, and Marcel Böhme. 2026. Empirical Computation: Prompting versus Programming. In *Proceedings of 41st IEEE/ACM International Conference on Automated Software Engineering (ASE'26 (NIER track)).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Model (LLM) agents are poised to change the field of software engineering. Only two years ago Microsoft announced the first LLM-based code synthesis tool Co-Pilot [15]. Now, *25% of all of Google's new code is LLM-generated* [7]—despite concerns about "hallucination" as a source of bugs and security flaws [16, 17].

Today, we solve computational problems[1] by *programming*: Using a programming language, we tell the machine precisely *how* instances of that problem are to be solved. For instance, if our problem is to sort an array of numbers, we think about a *specific* algorithm (e.g., merge sort) before developing a *specific* program (or function) that implements it. The program will expect the input to be given in a *specific* format, type, or data structure (e.g., `uint32_t`). A program thus "forces" the problem instance to be provided in the specified format. In software engineering, we have many approaches to analyze and even verify the correctness of such programs. We can localize and repair the underlying fault. Fundamentally, there exists no correct program that can solve the average instance of the sorting problem faster than $O(n \log n)$.

In the future, we expect *some* computational problems to be solved by *prompting* (e.g., using an informal encoding as a natural language prompt to an LLM or agentic system). The developer tells the agent informally in natural language about the instance of their computational problem, and the agent returns the result—if needed in a machine-readable format. The result is returned as empirically most likely rather than verifiably correct.

For our sorting problem, we can ask an LLM for instance to sort {一千一百二十三，九，负五，三点七}[2] and to return, e.g., a JSON file with the sorted array. In our experiments we find that the time it takes the LLM to solve sorting problems increases linearly in array size ($O(n)$ instead of $O(n \log n)$). That runtime increases at all is likely due to the increasing context length. Yet, we find that the likelihood that the LLM returns the sorted list *correctly* critically depends on the empirical likelihood of the problem instance:

- While the LLM sorts arrays of length 50 correctly in over 90% of cases, it sorts arrays of length 150 correctly only in 58% of cases.
- When *reasoning* (or "thinking") capability is enabled, LLM can achieve near-perfect correctness (within the time or token limits).
- While the LLM can sort the majority of arrays of length 20 correctly if the numbers are spelled out in English, it is far less reliable if numbers are spelled out in German.

We notice that, unlike for traditional programs, we cannot localize and repair the cause of an incorrectness once identified so that this and similar problem instances are solved correctly in the future. In the experiments for this vision paper, we also report on results of four other computational problems of varying complexity.

> *We argue that the capabilities and limits of empirical computation* cannot *be understood within the classic, rationalist framework of computation and call on the software engineering community to develop new instruments for the analysis of empirical computation.*

The purpose of this paper is to establish empirical computation as an emerging and interesting area in software engineering. What are the properties of empirical computation? How to analyze the fundamental limits? Are there any guarantees? How can we analyze, estimate, or predict the correctness of empirical computation in the general, in the problem-specific, or in the instance-specific?

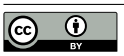
2 Motivation and Open Challenges

Programming vs prompting to solve problems. Today, we write programs to solve problems. As mentioned, a program represents *how* that problem is solved. Just like the problem can often be decomposed into smaller sub-problems, a program can be composed from many smaller computational units (e.g., functions). A program is a *specific* solution for a *specific* problem. The software engineering

*Both authors contributed equally. Eric conducted the work as intern at MPI-SP.

[1] In this paper, we take a very broad view of the term *computational problem*. For instance, the computational problem that is solved by a given program is the relationship between the inputs and outputs of that program (or the "purpose" of that program).

[2] The Chinese characters specify the array {1123, 9, -5, 3.7}.



ASE'26 (NIER track), 12–16 October 2026, Munich, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

community has developed many approaches and techniques to analyze the properties (incl. correctness) of a program [3]. However, in the future, we might simply prompt an agent to solve a problem. There is no program to analyze with respect to the problem it solves. **What can we say about the result? How do we even *test* the correctness of agents for various computational problems**?

Programming gives us absolute, direct, and precise control over the correctness and efficiency of a computation. The procedure of the computation is explicit and predefined. The fault of an incorrect computation can be localized and repaired. In contrast, prompting offers more flexibility, particularly when requirements are uncertain. Without constraints on language, input structure, or algorithm, an "empirical computer" returns the result after interpreting the user-provided, informal description of the problem statement using the available context. If we found the result to be incorrect, **how can we programmatically *improve* correctness for future instances** (without introducing regressions for other instances or problems)?

Formal vs informal representations of problem instances. In *programming*, inputs are provided using a *specific* format, type, or data structure. Everyone who, as part of a software testing class, has been asked to implement and to test a program for the triangle classification problem knows that we have to be *very precise* about that format. When parsing from a file, do we give lengths or angles, separated by commas or dashes? Are spaces and tabs allowed? When receiving as function input, should I use a primitive type, like (unsigned) integers, or implement my own class? Are negative numbers allowed? What about floating point values?

In *prompting*, the input can be provided informally in *any* format. The correct interpretation arises from context. Unlike in programming, there is no predefined "contract" between caller and callee that specifies the precise format or data structure for the input. This offers flexibility at the interface, but at the cost of ambiguity.

We experimentally explore the correctness of empirical sorting when numbers are provided not using digits but expressed entirely differently, e.g., using words in German (zweiundvierzig [speak, two-and-forty; 42]) or as characters in Korean. We find that the result of empirical sorting is more likely correct if numbers are provided in a language that is well-represented on the internet.

Limits in terms of efficiency vs correctness. The computational limits of programming to solve computational problems are well-studied in theoretical computer science. However, an LLM or agent finds answers to any computational problem in a time that aligns primarily with tokenization and runtime behavior rather than classical computational complexity. For instance, an LLM might take as much time to increment a number as it would to predict the plaintext from a given SHA-512 hash. Of course, correctness is a different matter. Are there problems that are particularly amenable to empirical computation? **How can we quantify the limits of empirical computation in the general or in the problem-specific**?[3]

We experimentally study the efficiency and correctness of *empirical computation* on simple, well-studied problems such as sorting and searching. In our measurements, runtime correlates primarily with tokenization and model/runtime behavior rather than classical problem complexity. For example, in *empirical sorting* or *empirical searching* (sorted or unsorted lists), the baseline model returns the correct result for lists of 50 random numbers in $\approx$ 90% of trials.

Formal vs informal notions of correctness. In programming, it is upon the programmer to elicit and to understand the computational problem to be solved (i.e., the requirements) before implementing a program that is meant to solve *all* instances of that problem. If the computational problem is precisely understood (such as sorting or searching), it can be encoded as formal specification, and given the specification, the programmer could formally verify, i.e., *guarantee* the correctness of her program w.r.t. that specification.

However, sometimes the problem is not precisely understood and can only be described vaguely (cf. requirements elicitation). In such cases, prompting offers more flexibility and allows a contextual interpretation of a vaguely described computational problem (which may even be further elicited interactively). Yet, unlike in formal language, a natural language description can be ambiguous, and the way the prompt is designed can substantially impact the correctness of the response (cf. prompt engineering). **What are effective ways to describe a computational problem to maximize correctness**?

3 Preliminary Experiments

To explore the challenges and opportunities of the empirical approach to computation, we conducted a large number of experiments. While we take a very broad view of the term "computational problem", here, for our experiments, we focus on simple, classic, well-studied problems, like sorting or searching, whose correctness is well-defined and computational complexity is well-studied.

3.1 Experimental Setup

Implementation. For our evaluation, we developed 1.2k lines of Python code that prompts an LLM for various computational problems like sorting, for which the concrete *prompt* is

> *"Sort the elements in the given collection in ascending order, and return only the sorted collection in the list format:* ⟨`numbers`⟩*."*

The input was provided as a Python list and formatted into a prompt string (`numbers`) according to problem-specific templates. The output was parsed from the LLM response into (hopefully sorted) Python list. In addition to sorting, we also conducted experiments on the following problems:

- searching sorted lists; $O(\log n)$,
- searching unsorted lists; $O(n)$,
- computing the longest palindromic substring; $O(n)$ [13], and
- finding a subset with a given subset sum; $O(2^{\frac{n}{2}})$ [9].

We chose to run our experiments with recent open-weights model `gemma-4-26b-a4b` [5] locally to eliminate network latency and ensure accurate timing measurements. The model ranks higher than `DeepSeek-R1-0528` [6] on the *Artificial Analysis Intelligence Index* [1] and is small enough to run on a MacBook Pro with Apple M5 Pro chip and 48GB unified memory. Based on the observed inference speed (~60 tokens/s), we set 600s timeout and 64k context length limit. We compare three configurations: *Default* (temperature=1.0), *Thinking* (reasoning mode enabled), and *Deterministic* (temperature=0.0).

[3] We note that PAC learning [21] and learnability theory offers insight only on the requirements for (or dependence on) the training data (i.e., sample complexity).

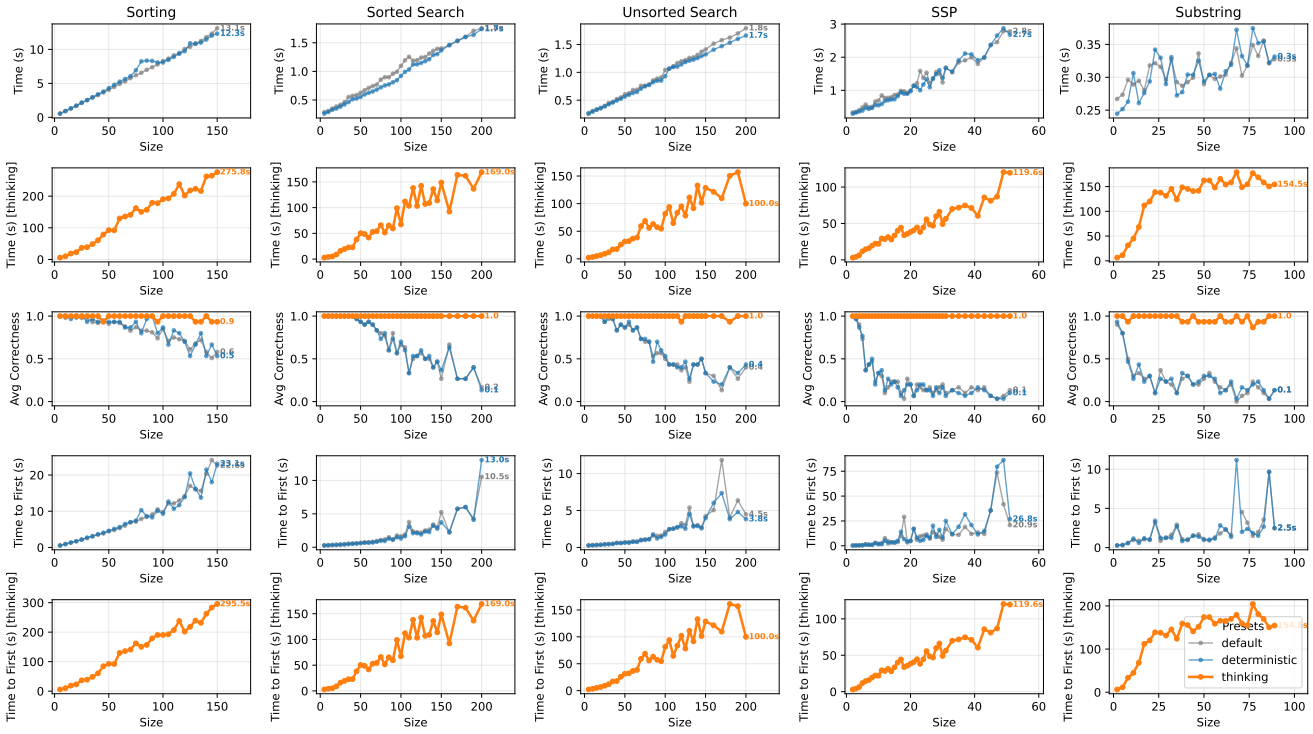


Figure 1: LLM-performance on various computational problems across *Default*, *Deterministic* (both at least 30 repetitions), and *Thinking* modes (15 repetitions). *Top*: Average time to solution. *Middle*: Average proportion of correct solutions. *Bottom*: Expected time to generate the first correct solution.

Methodology. For sorting, we generated Python lists of random numbers and of random length using existing functions from the standard Python library `random` (e.g., `random.sample`). As discussed in more detail in the individual sections, we varied properties like precision and magnitude of the numbers, as well as the length of the array. For every randomly generated input, we recorded the time taken (i.e., *efficiency*) and whether the LLM-generated result was correct (i.e., *correctness*).

For the other problems, we take a similar approach. For *searching*, we generate a random or sorted list of numbers as above, select a random element to search for and prompt the LLM to return the corresponding index or negative one (-1). For the subset sum problem (*SSP*), we generate a random list of numbers as above, find the sum S of a random subset, and prompt the LLM to return that subset whose total is S. For the longest palindromic substring problem (*substring*), we construct a random string of characters and prompt the LLM to return the result. For the latter two, we implement simple Python programs as ground truth.

3.2 Efficiency versus Correctness

Efficiency. Figure 1.top shows the efficiency of empirical computation, i.e., the average time it took the LLM to solve instances of various computational problems, as a function of instance size n. For four of five problems (except Substring) and all three modes, the time to find a solution is *linearly* increasing in list size. The increase is likely due to the corresponding increase in the number of input and output tokens [22], rather than the computational complexity of the problems. Note that the classic theory of computation predicts a difference between the average complexity of searching sorted versus unsorted lists. For Substring ($O(n)$), the execution time appears sublinear.

Correctness. Figure 1.middle shows the correctness of empirical computation, i.e., the proportion of empirically solved instances that are actually correct, as a function of n. Surprisingly, for *thinking* mode correctness remains high (>0.9) and unaffected by the size of the input for the sizes, we could reasonably test. For deterministic and default mode, we observe a clear decrease in correctness across problems. Interestingly, for sorting and both searching problems, the reduction in correctness in n is roughly equivalent: For Python lists with 50 integers, (a) the probability that empirical sorting returns the correctly sorted list is equivalent to a coin flip (0.5), and (b) the probability that empirical searching returns the correct index is also equivalent to a coin flip (0.5). In contrast, for the substring problem, correctness reduces to close to zero (0) already for strings with five characters likely due to tokenization. For the subset sum problem, correctness is close to zero (0) by lists of length 10. For these two modes, if we combine the efficiency and correctness results into the expected time until the LLM returns the first correct solution (Fig. 1.bottom), we can clearly see the consequence of the decrease in correctness on the performance of empirical computation for these problems.

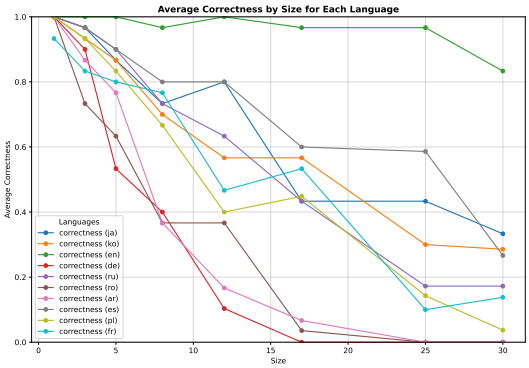


Figure 2: Correctness of empirical sorting across languages (i.e., the proportion of correctly sorted lists; default mode).

There might be several (non-traditional) mechanisms to improve the correctness of empirical computation, e.g., with methods like majority voting, prompt engineering [18], fine-tuning [8], reinforcement learning [10], or test-time-training [12]. Currently, for our sorting problem, the LLM would sometimes add numbers not present in the input or remove numbers that were present. Some numbers would appear multiple times, and sometimes the list would not be sorted entirely. Large inputs (e.g., $n > 200$) would sometimes be truncated or produce repeated sequences.

3.3 Inputs in Natural Language

A fundamental difference between the formal approach to computation and the empirical approach is that no "formal contract" is required for the latter on how the inputs are provided. In contrast, a program requires inputs to be passed in a certain structure that is formally agreed on beforehand. That input structure could be a specific file format, like this PDF file, a specific data structure, like a linked list, graph, tree, or object, or it could be a primitive data type, like a (signed or unsigned) integer or float.

In contrast, an LLM takes inputs flexibly, either informally in natural language or formally, e.g., as structured JSON file. The LLM infers from context how the provided input is interpreted. This flexibility enables broader applicability but also introduces ambiguities. There is no certainty about how an input is interpreted.

Methodology. We used a Python library called `num2words` to translate the random numbers in the list into words or characters in different languages. The list of languages is shown on the top right in Figure 2. We prompted the LLM (default mode) to sort the list of words and compared its output with the correct answer obtained by applying `num2words` to the sorted list of numbers.

Results. Figure 2 shows the correctness of empirical sorting, i.e., the proportion of empirically solved instances that are actually correct, as a function of n, when words or characters are used for numbers instead of digits. In comparison to Figure 1.middle, we can immediately see that the correctness of empirical sorting reduces more quickly. For languages that are well represented on the internet, correctness of empirical sorting is significantly better than for underrepresented languages. For instance, sorting a list of 20 numbers represented in English is correct with very high probability (>95%), while when represented in German it is almost never correct (<1%) in our 50 trials.

4 Perspective and Vision

*Everyone talks about trustworthy artificial intelligence. Yet, no one knows how to define, test, **improve**, or guarantee the correctness of the response of an LLM or an LLM agent to a given prompt. Our classic theory of computation offers no help in answering such questions; nor does the powerful program analysis machinery that we have developed in the software engineering community (e.g., static or dynamic analysis, verification, or model checking).*

*It is our **vision** that the analysis of the properties of empirical computation will emerge as a new area in software engineering that is both timely and rich with interesting problems.*

In the spirit of a research programme, we open several fundamental questions for the software engineering community.

Empirical program analysis. Given an empirical computer, like a system of LLM agents, what type of statements can we make about the correctness on various computational problems (whether or not formally specified [19])? How can we extend existing formal methods, like analysis or verification to work for empirical computers? In the absence of a program (i.e., a set of instructions) to analyze, we should develop tailored, scalable, blackbox empirical program analyses [2, 4, 11, 14] that proceed by statistical, counterfactual, or causal reasoning.

Prompt Engineering. What are the most effective encodings of a problem statement in natural language [23]? Can we predict the correctness for a given problem instance [20]? How can we (reactively—or better proactively) improve correctness on specific computational problems or their instances in the absence of a (problem-specific) program?

Limits and capabilities of empirical computation. How can we maximize the generality of our statements about the correctness of empirical computation? Can we make statements about correctness across *all instances* of a problem? Can we make general statements about the correctness of an empirical computer, i.e., across all problems, whether known or unknown? What are the fundamental limits of empirical computation?

Call to action. The analysis of correctness and other properties is a classic problem in software engineering. Hence, we call on the software engineering research community to develop the tools and techniques required to analyze the correctness of empirical computation as an important step toward the systematic analysis of the trustworthiness of artificial intelligence.

5 Data Availability

The implementation code, data, and scripts used in this study are available at https://github.com/chinggg/EmpiricalComputation.

Acknowledgments

This research was conducted as part of the CS@max planck internship program. This research is partially funded by the European Union (ERC project AT_SCALE, 101179366). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Artificial Analysis. 2026. LLM Leaderboard. https://artificialanalysis.ai/leaderboards/models Accessed on May 12, 2026.

[2] Marcel Böhme. 2022. Statistical Reasoning About Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, USA) *(ICSE 2022)*. 5 pages. https://doi.org/10.1145/3510455.3512796

[3] Marcel Böhme, Eric Bodden, Tevfik Bultan, Cristian Cadar, Yang Liu, and Giuseppe Scanniello. 2025. Software Security Analysis in 2030 and Beyond: A Research Roadmap. *ACM Transactions on Software Engineering and Methodology* (2025), 25 pages. https://doi.org/10.1145/3708533

[4] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 15th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 494–504. https://doi.org/10.1145/3468264.3468570

[5] Google DeepMind. 2026. Gemma 4: 26B A4B. https://huggingface.co/google/gemma-4-26B-A4B-it.

[6] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. https://arxiv.org/abs/2501.12948

[7] Benji Edwards. 2024. Google CEO says over 25% of new Google code is generated by AI. https://arstechnica.com/ai/2024/10/google-ceo-says-over-25-of-new-google-code-is-generated-by-ai/ (Accessed on 15/01/2025).

[8] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. arXiv:2403.14608 [cs.LG] https://arxiv.org/abs/2403.14608

[9] Ellis Horowitz and Sartaj Sahni. 1974. Computing Partitions with Applications to the Knapsack Problem. *Journal of the ACM* 21, 2 (1974), 277–292.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore. 1996. Reinforcement Learning: A Survey. arXiv:cs/9605103 [cs.AI] https://arxiv.org/abs/cs/9605103

[11] Seongmin Lee and Marcel Böhme. 2023. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. 12. https://doi.org/10.1145/3611643.3616268

[12] Jian Liang, Ran He, and Tieniu Tan. 2024. A Comprehensive Survey on Test-Time Adaptation Under Distribution Shifts. *International Journal of Computer Vision* 133, 1 (July 2024), 31–64. https://doi.org/10.1007/s11263-024-02181-w

[13] Glenn Manacher. 1975. A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM* 22, 3 (July 1975), 346–351. https://doi.org/10.1145/321892.321896

[14] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting information flow by mutating input data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 263–273.

[15] Microsoft. [n. d.]. Copilot. https://copilot.microsoft.com/. (Accessed on 15/01/2025).

[16] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *SP*. IEEE, 754–768.

[17] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. https://doi.org/10.1145/3576915.3623157

[18] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927 [cs.AI] https://arxiv.org/abs/2402.07927

[19] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. 2024. Specifications: The missing link to making the development of LLM systems an engineering discipline. arXiv:2412.05299 [cs.SE] https://arxiv.org/abs/2412.05299

[20] Thomas Valentin, Ardi Madadi, Gaetano Sapia, and Marcel Böhme. 2026. Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation. In *Proceedings of the 40th Annual AAAI Conference on Artificial Intelligence (AAAI'26)*. 14 pages.

[21] L. G. Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (Nov. 1984), 1134–1142. https://doi.org/10.1145/1968.1972

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903