AI-Assisted Programming in Practice Conceptual Foundations and Data-Informed Observations

Besiki Tabatadze

PhD (Applied Mathematics), Professor, European University, Georgian American University, Tbilisi, Georgia.

Email: *tabatadze.besik@eu.edu.ge*

Abstract

The increasing use of large language models in programming practice has transformed how software is designed, implemented, and validated. While existing studies largely focus on performance, productivity, or tool ecosystems, less attention has been paid to the role of requirement specification and interpretation in AI-assisted programming. This study examines how differences in task specification influence the behavior of humans and artificial intelligence during program construction.

The analysis is based on a comparative, practice-informed observation of programming tasks executed under two contrasting conditions: an incompletely specified task and a fully, explicitly specified task. Using a controlled programming context and the same execution environment, the study investigates how algorithmic assumptions, interpretative choices, and code structure emerge in each scenario for both human developers and AI-generated solutions. The focus is placed not on execution speed or optimality, but on interpretative variability, stability across executions, and qualitative properties of the resulting program code, including structural complexity and readability.

The observations show that incompletely specified tasks create a wide interpretative space in which AI-generated solutions exhibit substantial variability and, in some cases, exceed the intended scope of the task through functionally abundant or overly complex implementations. In contrast, explicit specification significantly reduces interpretative divergence and leads to conceptual convergence between human-written and AI-generated code. However, even under

fully specified conditions, differences remain in qualitative aspects of code structure, with AI-generated solutions tending toward higher structural complexity.

These findings highlight specification as a critical boundary condition in AI-assisted programming and emphasize the evolving role of humans from code production toward interpretation, validation, and refinement in modern programming workflows.

Keywords: AI-Assisted Programming, Large Language Models, Task Specification, Interpretive Differences, Program Synthesis, Human-in-the-Loop

Introduction

In recent years, the use of artificial intelligence in programming practice has expanded significantly. The development of large language models (LLMs) has made it possible to generate program code directly from natural language descriptions, whereby artificial intelligence no longer functions merely as a supporting tool but actively participates in the process of solving programming tasks (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022). Against this background, contemporary programming increasingly takes the form of a hybrid process in which human interpretation and machine-generated code coexist within a shared working environment.

In parallel with this trend, artificial intelligence is widely applied both in software systems and across various types of computational tasks, indicating its emergence as a general-purpose computational instrument rather than solely a means of automating specific operations. Research shows that the integration of AI technologies into system architectures and workflows substantially transforms the logic of task formulation, execution, and control, particularly in contexts where processes rely on textual, structural, or semantic interpretation and require human decision support (Chen et al., 2021; Tabatadze, 2024). At the same time, neural-network-based approaches are increasingly considered as alternative computational possibilities for

formally defined problems that have traditionally been addressed using strictly algorithmic or numerical methods, including programming tasks and scientific computing (Li et al., 2022; Tabatadze, 2025). A common characteristic of these trends is that the behavior of artificial intelligence depends fundamentally on the quality of task specification and on the constraints that define the boundaries of interpretation in both system-level and computational contexts.

In practical programming processes, tasks are rarely formulated through fully exhaustive and formal specifications. Developers frequently work with partially specified requirements, whose interpretation is grounded in experience, conventions, and contextual judgment. When such tasks are delegated to artificial intelligence, the level of specification becomes a decisive factor in shaping the resulting programmatic solution. Existing empirical studies indicate that incompletely specified tasks increase the variability of AI-generated code and lead to multiple possible implementations of the same task, differing both in internal logic and structural complexity (Gottlander & Khademi, 2023; Ramasamy, 2024; Chen et al., 2021).

In contemporary practice, this tendency is often reflected in prompt-based, exploratory programming approaches, where task formalization is minimal and outcomes depend strongly on interpretation (Ray, 2025; Taulli, 2024). While such approaches enhance flexibility and lower entry barriers, they simultaneously intensify interpretative uncertainty and complicate the assessment of stability and readability of the resulting programmatic solutions.

Research on code generation demonstrates that clear and detailed task specification reduces interpretative variability and promotes conceptual convergence among solutions (Li et al., 2022; Nijkamp et al., 2022). Nevertheless, even under fully specified conditions, AI-generated code frequently differs from human-written code in terms of structural organization, level of abstraction, and readability (Ray, 2025). These observations indicate that specification constitutes not merely a technical prerequisite but a critical boundary factor that governs interpretative freedom and the distribution of responsibility between humans and artificial intelligence within AI-assisted programming processes.

Within this context, the present study aims to examine how different levels of task specification influence the behavior of humans and artificial intelligence in the process of solving programming tasks. Through a practice-informed comparative analysis involving incompletely specified and fully specified formulations of the same task, the study focuses on interpretation, stability across repeated executions, and qualitative properties of program code, rather than on execution speed or algorithmic optimality. In doing so, the paper seeks to more clearly articulate the transformation of the human role in contemporary AI-assisted programming practice, from direct code authorship toward interpretation, validation, and simplification.

Conceptual Foundations of AI-Assisted Programming

AI-assisted programming represents a conceptual transformation in software development, within which program construction emerges from continuous interaction between human intent and probabilistic inference performed by machines. Unlike traditional programming paradigms, where algorithms are explicitly formulated, precisely specified, and directly implemented by human developers, AI-assisted programming relies on large language models that generate program code based on textual prompts, contextual constraints, and learned probability distributions. As a result, the programming process increasingly shifts from direct code creation toward the management, constraint, and interpretation of machine-generated solutions (Chen et al., 2021; Li et al., 2022).

One of the distinguishing characteristics of large language models is their reliance on probabilistic inference rather than deterministic rule execution. Code generated by AI does not constitute the execution of a formally defined algorithm, but rather a prediction of plausible continuations grounded in statistical regularities of code and natural language. Accordingly, code generation involves an interpretative process in which the model fills underspecified requirements, introduces implicit assumptions, and selects among multiple possible algorithmic realizations. This feature fundamentally differentiates AI-assisted programming from classical

automation tools and elevates specification from a secondary technical detail to a central conceptual element.

Within this context, task specification functions as a constraint space that defines the boundaries of interpretative freedom available to artificial intelligence. Incompletely specified tasks leave portions of this space unconstrained, allowing AI systems to introduce additional operations, abstractions, or structural generalizations that are not explicitly required by the task description. In contrast, clearly and explicitly defined specifications restrict the interpretative space and increase the likelihood of convergence toward a particular algorithmic structure (Nijkamp et al., 2022). Importantly, the relationship between specification and outcome is not binary; even minor changes in task formulation may lead to qualitatively different programmatic realizations, even when functional correctness is preserved.

Prompt-based programming practices have become increasingly widespread and follow a clearly positive trend. Vibe coding refers to an intuitive, loosely formalized interaction with AI systems, where priority is given to rapid iteration and outcome generation rather than detailed requirement formalization (Ray, 2025). While this approach lowers entry barriers and increases flexibility, it simultaneously amplifies interpretative variability and shifts responsibility toward the human participant as the final authority for evaluation and control (Taulli, 2024). At a conceptual level, this indicates that AI-assisted programming is not limited to accelerating code production, but also reshapes the internal cognitive and organizational structure of programming practice.

Similar tendencies can be observed in the broader application of artificial intelligence within software systems and computational tasks. The integration of AI technologies into system architectures and decision-support processes demonstrates that neural-network-based models increasingly operate within existing information systems, with their role in everyday practice steadily expanding (Tabatadze, 2024). Likewise, the application of deep learning in scientific computing shows that neural networks may be regarded as alternative solution paradigms even

for strictly formalized problems, further underscoring the role of specification across different computational contexts (Tabatadze, 2025).

Thus, AI-assisted programming can be understood as an interpretative process shaped by the interaction between task specification, probabilistic inference, and human oversight. The quality, structure, and stability of AI-generated code are determined not only by model capabilities, but also by how requirements and constraints are formulated. From this perspective, specification functions as a boundary condition that governs the distribution of roles between humans and artificial intelligence in contemporary AI-assisted programming workflows.

Research Methodology

The present study adopts a qualitative, practice-oriented methodological approach aimed at analyzing how different levels of task specification influence the behavior of humans and artificial intelligence in programming contexts. The study does not seek to evaluate model performance in terms of benchmarks, execution speed, or quantitative accuracy metrics; instead, it focuses on interpretative behavior as it emerges during the formation of programmatic solutions under varying conditions of requirement formalization.

The methodological design is comparative and observational. The same programming task is executed under two distinct conditions: an incompletely specified task and a fully, explicitly specified task. Both human developers and artificial intelligence operate within the same programming language and execution environment, ensuring that observed differences are primarily attributable to task specification and interpretative processes rather than technological factors. In this context, artificial intelligence is not treated as an experimental subject in a strict sense, but as an active participant in the programming workflow.

The study is grounded in programming practice rather than controlled experimentation. Its objective is not to test hypotheses related to optimal performance, but to observe how

assumptions, interpretations, and structural decisions emerge during the process of program construction. This practice-informed perspective makes it possible to analyze qualitative characteristics of program code, such as structural complexity, interpretative scope, and stability across repeated executions.

Interpretation is regarded as the central analytical component of the study. Programmatic solutions are evaluated in terms of how requirements are interpreted, which operations are included or excluded, and how consistently similar solutions are produced under identical task formulations. Particular attention is given to differences between human-written code and AI-generated code with respect to levels of abstraction, structural organization, and alignment with task requirements.

Data-Informed Observations from Programming Tasks with Different Levels of Specification

This section presents data-informed observations drawn from programming practice, focusing on the execution of programming tasks under different levels of specification, both with and without the use of artificial intelligence. The aim of the study is to examine how the roles of humans and artificial intelligence, the nature of interpretation, and the resulting outcomes in modern programming change depending on the degree of task formalization.

Two contrasting cases are examined:

1. A programming task that is incompletely specified and does not include all necessary details;
2. The same task formulated in a fully and explicitly specified manner.

The focus of the analysis is not on execution speed or code optimality. Instead, attention is directed toward the process of requirement interpretation: how it differs between humans and artificial intelligence, what kinds of interpretative assumptions emerge in each case, and how these assumptions affect the final outcome.

The task examined within the study belongs to the class of text data preprocessing (input preprocessing), which is widely used in programming practice and is conceptually simple. Nevertheless, tasks of this type are interpretatively sensitive, as their outcomes depend significantly on the level of requirement specification and the logic of interpretation applied during implementation.

To ensure the comparability of observations, the programmatic solution of the task was implemented using the same programming language and execution environment. The task was implemented in the Python programming language, version 3.11, which is widely established in text data processing practice. The choice of language is motivated by its high readability, its rich ecosystem for text manipulation, and the fact that it is equally common in both human-written and AI-generated program code.

In this context, the objective of the observation is not to evaluate specific syntactic features or language constructs. Python is used as a neutral medium that minimizes the influence of technological differences and allows the analysis to focus on the structural and conceptual properties of the programmatic solution.

Incompletely Specified Task: Interpretation and Outcomes

In the first research scenario, a deliberately incompletely specified programming task is employed, with the aim of revealing differences in requirement interpretation between humans and artificial intelligence when task specifications are provided at a minimal level. In both cases, task execution involves the creation of program code that automatically performs text normalization on the given data; manual text processing or editing is not part of the observation.

The programming task is defined as follows: given several paragraphs of textual data, develop a programmatic solution (an algorithm and the corresponding sourse code) that performs text normalization in such a way that the data become uniform and suitable for subsequent automated processing.

The task description does not define specific text normalization rules, applicable operations, or mechanisms for handling exceptions. The requirements do not specify how the processing of textual structural elements, special characters, whitespace, or letter case should be reflected in the program code; nor are formal criteria for evaluating the result provided. Consequently, the task does not prescribe a single unambiguous algorithmic standard and leaves room for multiple, equally plausible programmatic interpretations.

The textual material used for observation consists of paragraphs with differing structural characteristics, containing ordinary sentences as well as punctuation and ellipses, mixed letter case (e.g., combinations of upper- and lower-case letters), redundant whitespace, and, in some cases, special characters and digits. This heterogeneity implies that textual "uniformity" can be achieved through different algorithmic strategies, and the choice of a particular strategy is reflected in the program code produced by the executor.

Such incomplete formalization creates an interpretative space in which the programmatic solution depends on the assumptions and design decisions implemented in the code. For this reason, the unit of comparative analysis in this scenario is not execution speed or syntactic style, but rather which algorithmic assumptions are reflected in the code during the definition of the normalization process and how stably or variably these assumptions manifest in programmatic solutions generated by humans and artificial intelligence.

Accordingly, the evaluation is conducted based on the conceptual profile of the solutions, which involves identifying the following aspects:

1. which operations are implemented in the program code as essential components of normalization (e.g., case normalization, whitespace consolidation, handling of punctuation or special characters);
2. how the processing of textual structural elements is reflected in the code (preservation or merging of paragraphs and lines);

3. how exceptions are handled within the algorithmic logic (e.g., empty lines, non-standard characters, mixed formats);
4. how consistent the program logic generated by the same executor remains under repeated executions.

This analytical framework allows human-created programmatic solutions to be interpreted as experience-based and relatively stable algorithmic interpretations, whereas program code generated by artificial intelligence can be viewed as a spectrum of possible programmatic realizations of the same requirements, which may vary even when the task formulation itself remains unchanged.

The theoretical considerations are further reinforced by empirical observations. When humans construct a programmatic solution, a clear and coherent interpretative framework emerges, grounded in commonly accepted though informal practices of text normalization. Within the program logic, normalization is understood not as a transformation of textual content, but as a form of formal stabilization aimed at ensuring a uniform representation suitable for subsequent automated processing.

More specifically, the programmatic strategy adopted by humans is implemented through operations that standardize the formal characteristics of the text (such as letter case, whitespace, and structural boundaries), while deliberately avoiding algorithmic steps that would alter the content level of the text or the form of individual words. This choice indicates that normalization, as encoded in the program logic, is defined as a minimal yet sufficient transformation.

Observation of task execution by artificial intelligence reveals a different pattern. Under the same task formulation, programmatic solutions generated by artificial intelligence differ from one another in terms of the number of normalization operations implemented in the code, as well as in their content, ordering, and interpretative scope. The observed generations do not converge toward a single algorithmic interpretative framework and, in some cases, extend beyond the boundaries of formal normalization as a purely technical task.

More specifically, in some generations artificial intelligence interprets normalization at the programmatic level as a minimal technical operation, and the corresponding code implements only those steps required for superficial textual stabilization. In other generations, however, the generated code includes additional algorithmic operations that exert a content-level influence on the processed text. Such changes manifest in the program logic through alterations in word order within sentences, reformulation of expressions, the use of synonym substitutions, as well as structural simplification or merging of sentences. Although these transformations often preserve the general meaning of the text, they modify its formulation and semantic emphasis and therefore exceed the scope of normalization understood as formal standardization.

It is particularly important that these content-related effects emerge even in cases where the task description does not require text rewriting or reformulation. This indicates that, during the process of program code generation, artificial intelligence supplements the task requirements based on its internal patterns and language modeling mechanisms. As a result, normalization in some generations algorithmically transforms into a process of text reformulation, increasing the risk of content distortion and reducing the predictability of the resulting programmatic solution.

This interpretative variability is directly reflected in the structure of the generated program code. Observations show that, despite functional correctness, code generated by artificial intelligence often exhibits relatively high structural complexity. Such code frequently introduces multiple logical layers, additional helper constructs, and generalized structures that are not strictly required by the task context but emerge as a consequence of an expanded interpretation of the requirements.

Consequently, reading and comprehending the generated program code requires greater effort, particularly when the task itself is incompletely specified and interpretative assumptions are already substantial. The internal logic of the code is often not directly aligned with the concise task description, increasing the cognitive load during subsequent analysis and validation. From this perspective, comparison is no longer limited to whether the code fulfills the required

function, but also encompasses code comprehensibility namely, how easily it can be read, understood, and modified by a third party.

Overall, the observations indicate that AI-generated programmatic solutions frequently exhibit an approach oriented toward so-called “functional abundance,” which, in the context of tasks of limited scope, results in excessive structural complexity. This phenomenon is particularly pronounced in incompletely specified tasks, where interpretation is already challenging. Under such conditions, overly complex and difficult-to-read program code reduces the practical effectiveness of AI-assisted programming and increases the role of the human not only in interpretation, but also in understanding, refactoring, and simplifying the generated code.

Fully (Explicitly) Specified Task: Interpretation and Outcomes

In the second research scenario, the same programming task is presented in a fully and explicitly specified form, with the aim of minimizing interpretative freedom and clearly defining the required operations, their order, assumptions, and constraints. In this scenario, the task formulation constitutes an unambiguous algorithmic specification within which the programmatic solution must be constructed.

The following programming task is defined (Fully Specified Programming Task):

Several paragraphs of textual data are given (either a list of strings or one large text). Write a program or a function normalize_text(paragraphs) that applies the normalization steps listed below to each paragraph in the exact order specified and returns the normalized paragraphs in the same order.

The input may be:

list[str] — a list of paragraphs; or str — a single text in which paragraphs are separated by a blank line.

The output must be list[str] — a list of normalized paragraphs. Paragraphs that become empty after normalization must not be included in the output.

The program must implement the following normalization steps in the exact order given:

1. Remove leading and trailing whitespace from each paragraph;
2. Convert any repeated whitespace into a single space;
3. Convert the text to lowercase;
4. Apply Unicode normalization using the NFKC form;
5. Normalize punctuation (standardize dashes and ellipses);
6. Normalize quotation marks;
7. Remove control characters;
8. Ensure exactly one space after punctuation marks;
9. Remove empty paragraphs.

In addition, the following constraints are specified to rule out content-level transformation of the text:

1. Do not change the order of words;
2. Do not change word forms;
3. Do not remove numbers or delete punctuation (except for the normalization defined above);
4. The programmatic solution must be idempotent, meaning that repeated execution must not change the result.

Under conditions where the task is fully and explicitly specified, the development of a programmatic solution by humans primarily constitutes a step-by-step implementation of the stated requirements. The space for interpretative choice is reduced to a minimum, as each operation is defined both in terms of content and exact execution order, while the stated constraints further delimit the task. In this context, the human working process is oriented toward tracing the specification: steps are followed sequentially, such that each code block can

be directly mapped to a corresponding item in the task description, and subsequent verification (e.g., compliance with idempotency or the rule of removing empty paragraphs) can be performed through straightforward logical checks.

In human-written program code, a minimalist structure typically emerges. A single main function is implemented, supplemented, if necessary, by a limited number of helper operations that serve the realization of the explicitly listed rules (such as Unicode NFKC normalization, consolidation of whitespace into a single space, or punctuation standardization). This structure enhances code transparency, reduces unnecessary generalization, and simplifies later modification or testing, as each requirement of the specification is represented in the code in a visible and localized manner.

Notably, in this scenario, solutions produced by humans exhibit a high degree of consistency under repeated execution. Detailed requirements establish a stable framework within which the code is less susceptible to variation: any differences that do arise are typically limited to stylistic aspects (such as the organization of function blocks or the use of helper functions), while the functional content and the order of operations remain virtually unchanged.

Observation of task execution by artificial intelligence under fully specified conditions shows that a clear specification significantly reduces interpretative variability. Program code generated by AI generally implements the required steps correctly and, under the given constraints, avoids content-level transformations of the text (such as reordering words or altering word forms). Moreover, explicitly defined test criteria—most notably idempotency—act as a critical boundary condition that enables AI to verify logical consistency and detect obvious conflicts among the rules.

Nevertheless, despite functional correctness, AI-generated program code often continues to exhibit relatively high structural complexity. Observations indicate that AI frequently produces a more builder-style solution, introducing additional helper functions, generalized configurations, or redundant logical branches “for safety,” even when the specification does not

explicitly require them. As a result, the code may be functionally correct and idempotent, yet its readability and rapid validation demand greater cognitive effort than that of human-written code.

In the fully specified task scenario, comparative analysis demonstrates that detailed specification ensures conceptual convergence of programmatic solutions: code produced by both humans and artificial intelligence is oriented toward the precise execution of the same rules, and interpretative differences are reduced to a minimum. Under these conditions, artificial intelligence exhibits a clear advantage in the operational dimension of task execution, rapidly generating functionally correct and specification-compliant solutions that satisfy the idempotency requirement.

At the same time, differences remain clearly visible in qualitative aspects of code. Human-created programmatic solutions are typically shorter, more direct, and easier to read, whereas AI-generated code often includes additional structural layers and helper constructs. Consequently, in fully specified tasks, the role of the human shifts away from interpretation toward the evaluation, simplification, and optimization of the generated code.

Conclusion

This paper has examined AI-assisted programming as an interpretative and collaborative process in which the formation of programmatic solutions depends not only on the capabilities of the applied model, but also on the quality and logic of task specification. The study demonstrates that the integration of large language models into programming practice fundamentally transforms the process of program construction, shifting the focus from direct code writing toward the interpretation of requirements, the formulation of constraints, and the evaluation of generated outcomes.

Practice-informed observations indicate that incompletely specified tasks create a broad interpretative space in which AI-generated programmatic solutions are characterized by high

variability, functional abundance, and frequently excessive structural complexity. Under such conditions, artificial intelligence tends to supplement task requirements with implicit assumptions, increasing the risk of semantic deviation and reducing code readability. In contrast, human-developed solutions in the same context typically rely on more stable interpretative frameworks and exhibit clearer, more comprehensible algorithmic logic.

In cases where tasks are fully and explicitly specified, the observations reveal a conceptual convergence between solutions produced by humans and artificial intelligence. Detailed specification significantly reduces interpretative divergence and ensures a high level of functional alignment. Nevertheless, even under these conditions, AI-generated code often retains relatively high structural complexity, while human-written code is more strongly oriented toward evaluation, simplification, and optimization.

Based on these findings, task specification in AI-assisted programming can be understood not merely as a technical requirement, but as a boundary condition that defines the limits of interpretative freedom and shapes the distribution of roles between humans and artificial intelligence in the programming process. In contemporary practice, the human role is increasingly less confined to direct code authorship and more focused on interpretation, validation, and responsibility for the resulting programmatic solutions.

Thus, AI-assisted programming should not be viewed as a direct replacement for traditional programming, but rather as a new conceptual framework in which programmatic outcomes emerge through dynamic interaction between human judgment and probabilistic machine inference. The study underscores that the quality of this interaction ultimately depends on how clearly, consistently, and thoughtfully the task is formulated, and on the degree to which its specification is detailed and complete.

REFERENCES

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde de Oliveira Pinto, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., … Zaremba, W. (2021). *Evaluating large language models trained on code*. *arXiv*. https://doi.org/10.48550/arXiv.2107.03374

Fan, G., Liu, D., Zhang, R., & Pan, L. (2025). The impact of AI-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches. *International Journal of STEM Education*, *12*(1), 16.

Gottlander, J., & Khademi, T. (2023). The Effects of AI Assisted Programming in Software Engineering.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with alphacode. *Science*, *378*(6624), 1092-1097.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., ... & Xiong, C. (2022). Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Ramasamy, D. (2024). *Towards AI-Assisted Data Science Development: Decoding, Visualising, and Enhancing Human-AI Collaboration for Data Science Workflows in Practice* (Doctoral dissertation, University of Zurich).

Ray, P. P. (2025). A Review on Vibe Coding: Fundamentals, State-of-the-art, Challenges and Future Directions. *Authorea Preprints*.

Tabatadze, B. (2024). Prospects of Using AI Technologies in Open Journal Systems (OJS). *Journal of Technical Science and Technologies*, *8*(2), 68-74.

Tabatadze, B. (2025). Solving partial differential equations with deep neural networks. *In Proceedings of the International Conference on Global Practice of Multidisciplinary Scientific Studies II.* 763.

Taulli, T. (2024). *AI-assisted programming: Better planning, coding, testing, and deployment*. O'Reilly Media, Inc..

Wang, Y., Yu, P. D., & Tan, C. W. (2025). Future-Proofing Programmers: Optimal Knowledge Tracing for AI-Assisted Personalized Education. *arXiv preprint arXiv:2509.23996*.