The Modular Imperative: Rethinking LLMs for Maintainable Software

Anastasiya Kravchuk-Kirilyuk
Harvard University
Cambridge, MA, USA
akravchukkirilyuk@g.harvard.edu

Fernanda Graciolli
Midspiral
Cambridge, MA, USA
fernanda@midspiral.com

Nada Amin
Harvard University
Cambridge, MA, USA
namin@seas.harvard.edu

Abstract

Large language models (LLMs) are becoming increasingly integrated into software development, with a majority of developers now adopting AI tools for code generation. Although the current models can often produce syntactically and functionally correct code, they often generate unnecessarily complex solutions, and struggle with large, evolving code bases that have rich internal structure. Most evaluations of LLM-generated code to date have focused primarily on test-based accuracy, unfairly overlooking other essential aspects of software quality. In this paper, we emphasize the importance of modularity — the practice of structuring code into well-defined, reusable components — as a critical lens for improving the maintainability of AI-generated code. We argue that modularity should be a foundational principle in LLM-assisted code generation, empowering models to produce more maintainable, production-ready software.

CCS Concepts: • **Software and its engineering** → *Abstraction, modeling and modularity*; *Software design techniques*; *Reusability*; *Source code generation*; *Automatic programming*; • **Computing methodologies** → *Machine learning approaches*; *Natural language processing*; • **Social and professional topics** → *Software maintenance.*

Keywords: maintainability, modularity, LLM, software

ACM Reference Format:
Anastasiya Kravchuk-Kirilyuk, Fernanda Graciolli, and Nada Amin. 2025. The Modular Imperative: Rethinking LLMs for Maintainable Software. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (LMPL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759425.3763392

1 Introduction

Large language models (LLMs) are rapidly becoming more adept at code generation. The latest models can often generate syntactically and functionally correct code, although typically without formal guarantees, and have thus become ubiquitous among programmers for discharging tasks. A recent StackOverflow survey shows that 76% of developers use or plan to use AI coding tools [5].

However, while LLMs excel at generating code-based solutions in isolation, software development as a whole requires more than appropriate syntax and test-based accuracy. Large-scale, maintainable software requires careful attention to scope control, architectural consistency, and code simplicity (among other best practices). Yet LLMs fall short in precisely these areas: they often generate solutions that include extraneous functionality, struggle to maintain consistency across files, and produce verbose implementations where simpler solutions would suffice.

While all of these limitations impact code quality and developer experience, in this paper we focus on one crucial limitation: LLMs' failure to understand and respect *modularity* — a fundamental organizing principle that enables software systems to scale without sacrificing long-term maintainability. We demonstrate how this failure manifests both when generating new systems and when modifying existing code bases, and argue that addressing modularity represents a critical next step for making AI-generated code truly production-ready.

2 Why Modularity?

Modularity is a foundational principle in software engineering that promotes the decomposition of complex systems into smaller, self-contained organizational units. Each such unit represents a specific functionality and can interact with the rest of the code base via an interface. This promotes logical organization of related code, careful dependency analysis on the part of the programmer, and code reuse. In practice, modularity can be defined at various levels of granularity, from singular functions, to classes, modules, libraries, and even entire applications. Most programming languages support modular programming paradigms; it is simply a matter of consciously using them.

Modular design is held in high regard due to its significant benefits for code production and maintenance. In large evolving code bases, modularity can increase production as different modules can be built independently and concurrently by team members. By supporting the separation of concerns,

LMPL '25, Singapore, Singapore

ACM ISBN 979-8-4007-2148-9/25/10
https://doi.org/10.1145/3759425.3763392

modularity allows developers to make contributions to targeted functionality without needing to understand the entire code base. Modularity decentralizes debugging, as components can be tested independently and in conjunction with the rest of the system. Finally, modular design can improve code comprehension and documentation — an important factor for long-term maintainability.

3 Position

Due to these benefits, modularity plays a critical role in producing robust, scalable, and maintainable software. This is why we believe it should be a key guiding principle for LLM-assisted code generation. It is not enough for LLM-generated code to *appear modular* by splitting functionality across multiple files or modules. Effective modularity requires coherent boundaries, reusable abstractions, and minimal coupling: principles that current LLM outputs often violate, despite the surface structure. We believe that adherence to *modularity principles* should be built into LLM-synthesized code by default, supporting targeted, incremental refinement as the code base evolves. In practice, this means LLMs must not only generate modular structures that respect principles, but also reason effectively about existing modular components, helping programmers manage the complexity of evolving code through edits, debugging, and test generation.

Current LLMs optimize for immediate completeness and accuracy over architectural integrity and long-term maintainability. When given freedom over a full code base, LLMs tend toward comprehensive (often over-engineered), boundary-crossing solutions that may be functional on the surface, but render the underlying code brittle. This approach undermines effective modularity, compelling programmers to resort to ad hoc strategies in an attempt to enforce it on LLM-generated code. For example, when constrained to specific modules, via prompting or careful scaffolding (e.g., an intermediate language layer), LLMs are often able to produce appropriate, focused solutions that maintain long-term structural integrity. However, both prompting and scaffolding are external strategies that place the burden of constraining the LLM onto the programmer, while lacking consistent or reproducible results. We believe that a more integrated approach to modularity would empower LLMs to internalize such constraints, helping LLMs generate principled code with minimal intervention from the programmer.

4 The State of Affairs

How close are we to fully leveraging modularity in LLM-synthesized code? Our experiments suggest that modular programming with LLMs still falls short across various aspects. We ran multiple experiments, some with the help of Rubric DSL.[1] Rubric is specifically designed to work with AI. Its core purpose is not to generate programs by itself, but to constrain, guide, and validate code generation and code base architecture. It acts as a "semantic contract" layer between natural language prompts and the AI output. Running our experiments with the addition of Rubric alongside a baseline LLM illustrates that LLMs alone are not currently capable of generating proper architectural organization from only high-level natural language prompts. Although the addition of Rubric did not always lead to architectural excellence (this was expected as Rubric is in early stages of development), its overall positive impact points to a promising direction: external constraint layers like Rubric can begin to encode and enforce modular design principles that LLMs currently fail to internalize, offering a path toward more principled and maintainable AI-assisted software development.

Our experiments show that even with modular guidelines — such as specification files, existing modular structure in the code, or targeted prompting — the LLM often produced code that appeared modular at a glance, but still violated the principle through over-engineering, hidden dependencies, and breaking changes.[2] We discuss our insights below.

4.1 Modular Code from Scratch

As suggested by the existing studies in this area, LLMs can produce modular code from scratch *when prompted to do so* (whether via prompt decomposition, sketching and scaffolding, or demonstrating modular reasoning during training), and can even see performance gains on benchmarks as a result [2, 3, 6, 8, 9]. Inspired by this related work, our first goal was to examine whether an LLM would naturally produce modular code (without specification guidance), and how explicit modular specifications would affect the generated architecture. To this end, we ran an experiment using Cursor with Claude Sonnet 4 in which the coding task was to generate an HTML / CSS / JavaScript mock blog dashboard. Without any modular guidance, the LLM produced a monolithic solution with all functionality in a single file (586 lines). When provided with a Rubric specification defining modular structure, the same prompt yielded a code base that exhibited surface-level modularity — separate files for each component, consistent interfaces, event-based communication, and reusable design tokens. However, it still suffered from hidden dependencies between components, duplicated mock data that required manual synchronization, and excessive complexity. Despite the 4x increase in code size, the added complexity did not translate to any additional functionality. The outcome of this experiment suggests that, even with specification guidance, LLMs can still underperform on the facets of modularity that facilitate long-term maintainability.

[1] Rubric DSL is available at https://rubric.midspiral.com

[2] The code from our experiments, along with instructions on how to reproduce them, is available at https://github.com/graciolli-f/Modularity-Paper-Experiments

4.2 Working with Modular Code

4.2.1 Porting Code Base to React. Using the previously generated dashboard code bases (monolithic and modular), we tested the LLM's ability to port the HTML / CSS / JavaScript code to React. We expected that the modular code base would be ported more effectively since its separation of concerns maps naturally to React components. However, the LLM introduced several bugs to the modular code, while the monolithic code base ported cleanly in one shot. Although the LLM was able to catch and fix most bugs introduced in the modular code base during the porting process (i.e., before surfacing the result to the user), one bug remained which required five additional debugging prompts, gradually increasing in specificity, to pinpoint and fix it.

Our impression from this experiment was that the LLM tried to maintain — or, perhaps more accurately, mimic — the current code base's loose architectural philosophy, but without adhering to any specific principle. So, the monolithic code base was ported to React with minimal restructuring: it maintained global functions and passed props directly. While it was a clean port, the code base itself would not be suitable for production. On the other hand, the more modular code base led the LLM to introduce unnecessary complexity: over-engineered React-specific optimizations (e.g., useMemo, useCallback, refs) that were not necessary given the original implementation, and circular dependency issues. Rather than recognizing that React's built-in component model already provides modularity, the LLM seemed to amplify the complexity signals it detected in the source code.

While this test provides an interesting comparison of the LLM's approach to a monolithic vs. modular code base, we need to consider that the React framework uses a specific approach to modularity and it's possible that the existing loosely modular organization did not match React's patterns making it more difficult to re-structure and port.

4.2.2 Targeted Edits. Lastly, we tested the LLM's ability to maintain modularity when making edits. For this, we used a very simple modular Todo app with about 100 lines of code. Each module had one responsibility, there were clean interfaces between modules, and we introduced private members. We wanted to test the LLM's targeted edits capabilities at two levels: intra- and inter-module. Our test suite consisted of four experiments with increasing architectural complexity. Each experiment was performed with and without a Rubric specification in the directory, to compare the baseline capacity of the LLM to perform these tasks to its capacity in a setting with some structural support. We present the aggregated results from two independent runs below.

Experiment 1: Simple intra-module edit. We asked the LLM to *"Add priority field validation to the Todo model."* This request clearly scoped to a single module. Without Rubric, the LLM modified all four files in one of the runs – demonstrating it may not neccessarily respect boundaries even when the task is unambiguous. When using a Rubric specification, only `todo.js` was modified as expected.

Experiment 2: Complex intra-module edit. We asked the LLM to *"Make the store support filtering todos by any field dynamically."* Despite this being a single store-level feature, the LLM modified multiple files, and most notably, turned `App.js` into a test showcase in both runs. In one of the runs, the LLM also added a complex query DSL and modified the renderer. Including a Rubric specification ensured that only `store.js` was modified.

Experiment 2a: Add constraints. To explore further and find the limits of the baseline capacity of the LLM, we introduced two progressively more strict constraints to the prompt, leading the LLM to finally generate a clean, targeted edit solution. First, we appended a simple directive to the original prompt: *"Respect modularity."* With this, the LLM slightly reduced the complexity of its solution and modified fewer files, but still over-engineered by adding tests and modifying the demo. Next, we appended *"Only implement what is explicitly requested."* With this constraint, the LLM was able to produce an appropriately scoped solution: modification to one file, clean implementation, and a simple API. Here we see that while LLMs are capable of respecting module boundaries and maintaining architectural integrity, they default to over-engineered solutions, unless defensive prompting is included in the request. We believe this default tendency should be reversed with a stronger bias for simpler solutions constrained to modules, unless explicitly instructed otherwise.

Experiment 3: Architectural recognition. For this test we requested a feature without providing implementation guidance: *"Add functionality to get statistics about todos."* The goal was to determine whether the baseline LLM would recognize the existing modular architecture of the code base, and implement the new feature in a principled, modular fashion. It did not; instead of creating a new uniquely responsible module, it added the feature to the `TodoStore`. It also added edits to multiple other files, and, most critically, introduced a breaking change by modifying the `add()` method signature in one of the runs. When provided with a Rubric specification, the LLM did successfully implement the feature in a new module.

Experiment 4: Module integration. Lastly, we tested the LLM's ability to create a new module when explicitly instructed to do so, and then integrate it into the existing system: *"Add a new stats module that counts todos by status."* While the LLM was able to generate a proper, clean `stats.js` module, it then violated encapsulation in one of the runs by directly accessing private store members

(`_nextId` and `_todos`) instead of using the available public API. With Rubric, no such violations were witnessed.

This encapsulation violation directly undermines modularity. By accessing private members, the LLM created hidden dependencies between modules, making them no longer truly modular. The LLM generated code with the *appearance* of modularity (separate files, dependency injection) while violating the very principles that make modularity valuable. This is evidence of architectural mimicry without adherence to the underlying architectural principles.

5 Discussion

Our experiments reveal a fundamental challenge with the current state of LLM-assisted programming: while modularity is essential for building maintainable production software, LLMs consistently violate modular principles by default. A lot of emphasis (training, research, benchmarking) has been placed on accuracy, but even a perfectly correct function that ignores established interfaces, for example, undermines the integrity of the very code base it was tasked with improving. Here we discuss the implications of our findings and explore potential paths forward.

5.1 Encoding Principles

In our experiments, we repeatedly observed the LLM treating modularity as a pattern to mimic, rather than an underlying principle for decision-making. This gap between pattern and principle was illustrated in our module integration experiment, where the LLM chose the path for creating a new module that violated existing boundaries, despite there already being an established path that respected boundaries (public API). Humans have the ability to create a mental representation of "modular design" and make implicit decisions based on this representation. How can we similarly encode principles — not mere patterns — into the LLM's decision-making process? We outline a few potential approaches below.

Modularity Training. If modularity is to become a guiding principle in LLM code generation, this emphasis must begin with training. Just as careful planning improves the quality of execution, targeted training can cultivate modularity, encouraging models to decompose problems thoughtfully before attempting solutions. To accomplish this, models can be trained to have a stronger bias toward respecting modular boundaries rather than complete solutions — for example, training on examples where restraint is rewarded, and more resources are allotted to task breakdown and mapping. We must also train models to prioritize minimal, conservative edits over sweeping changes across components.

Built-in Metrics. LLMs excel at following concrete rules (e.g., "must not access any property or method from another module that starts with _"), but struggle to evaluate more abstract qualities (e.g., "avoid tight coupling"). We could lean on the LLM's strengths and use concrete metrics (e.g., the number of external dependencies of a module) and unambiguous rules (e.g., "must not introduce any new dependencies to this module") that serve as a proxy for modular reasoning. This approach, especially if implemented via a DSL like Rubric, also opens the door to runtime validation which can flag boundary violations that can be fed back to the LLM for real-time adjustments. Ultimately, while metrics and rules are not internal to the LLM itself, they can be part of the validation strategy built into the broader tooling and developer workflow.

Structured Support. To support the LLM in generating quality modular code, we can set up structured scaffolding that acts as the LLM's internal representation of the modular principles. The LLM should be able to refer to this scaffolding when making decisions during code generation. This structured scaffolding could manifest in different ways. For example, the LLM could keep an internal network of log files, tracking the architecture of the repository, the concepts represented by components, and any feature edits introduced over time. The LLM could automatically generate and consider high-level artifacts, such as dependency graphs, as part of its context alongside code. Structure could also be enforced by employing verification or static analysis techniques, such as code flow analysis (CFA) or constraint checking during code generation, serving as a safeguard against boundary breaking code.

Flexibility. One important limitation of modularity is that, realistically, the optimal modular breakdown will be project- and language-specific, and we cannot pick a single approach in advance. This is evidenced by our porting experiment, where approaches to modularity differed in the source and target, resulting in a suboptimal port. Furthermore, changing requirements over time may alter any existing modular structure. With this limitation in mind, modular code generation via LLMs should be approached as an evolving design and refactoring process, and not as a fixed solution.

5.2 Specialized Benchmarking

Benchmarking LLM performance with respect to modularity presents unique challenges. Most existing benchmarks are designed to assess code accuracy, and contain single-file programs that tackle isolated tasks. As such, they are not sufficient to evaluate the ability of LLMs to produce modular, maintainable, and scalable code. Measuring modularity is more nebulous than measuring correctness: it is a multidimensional property that depends on principles like separation of concerns, clarity of APIs, and code reuse. There is usually no single "correct" breakdown of a program into modular components, making ground truth comparison infeasible. Additionally, the benefits of modularity often emerge over time in refactoring or debugging, and are difficult to

capture in static benchmarks. Recent work has begun to move the needle in this direction, with Zhong and Wang [10] presenting a benchmark for evaluating the reliability and robustness of LLM-generated code, focusing mainly on API misuses. While this is a great step toward a more robust assessment, there is still work to be done to support an empirical assessment of modularity. New benchmarks that attempt to bridge this gap could feature both monolithic and modular versions of the same code, supporting direct comparisons of design quality and maintainability. For a more comprehensive assessment of modular design, benchmarks could include projects with complex component structure and dependencies across files that better reflect real-world software frameworks. While a single metric cannot capture modularity, there are many ways to quantify modular *qualities* of code – from established metrics like coupling and cohesion, to customized project-related criteria. The priority is not to achieve flawless modular design, but to recognize and assess modularity as a key quality of LLM-generated code.

6 Related Work

The difficulty in building robust and maintainable systems has long been recognized. In his seminal essay, Brooks [1] argues that the complexities of building and maintaining software lie in the specification, design, and testing of interlocking concepts. One difficulty he points out is changeability: software evolves over time due to new applications and extensions, and due to outliving its hardware. Brooks mentions that modular design may reduce the labor costs of generating test suites, as well as help with long-term maintenance and testing of the evolving code base. Brooks also argues that the difficulty of creating conceptually complex code can be mitigated by reusing already built software components. These sentiments still remain relevant today.

Work in LLM-assisted program synthesis has embraced decomposition as a mechanism for improving reasoning and the quality of generated code. For example, Parsel [9] decomposes implementation tasks into strongly connected components, solutions to which are then synthesized automatically. Similarly, ANPL [2] decomposes a programming task into a dataflow sketch that provides structure, and natural language holes that can be implemented or decomposed further. Other methods rely more heavily on prompting: the DeAR framework [8] uses question decomposition to build a reasoning tree, solves sub-questions at each node, and propagates the information back up the tree for a final composite answer at the root. Chain-of-Thought prompting [6] relies on demonstrating intermediate reasoning steps during prompting to elicit similarly structured multi-step responses. Others focus on training: Jain et al. [3] investigate the benefits of LLM-assisted modularization for training more accurate code generators, seeing gains in benchmark performance by up to 30%. Other works focus on decomposition-aided debugging. For example, Wen et al. [7] explore decomposition as a way to assist human programmers in debugging synthesized code, where each solution is broken down into smaller pieces that are easier for the human to repair. Shi et al. [4] use decomposition as an aid in automated debugging by breaking code down into a tree of sub-functions and resolving bugs at different levels of granularity (corresponding to different levels in the tree). These results imply that code modularity can be helpful for both humans and automated tools in isolating bugs.

7 Conclusion

As LLMs continue advancing at code generation and developers embrace them as part of their workflow, it is no longer sufficient to focus solely on the accuracy of the generated code. To ensure that LLM-generated code can reliably evolve through incremental development, and remain maintainable over time in production environments, modularity should become a guiding principle for LLM code generation. By deeply leveraging modular design rather than mimicking modularity, LLMs can generate code that is easier to debug, extend, and integrate. Embracing modularity is a key next step to advance LLMs from lump code generators to effective collaborators.

References

[1] Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20, 4 (1987), 10–19. doi:10.1109/MC.1987.1663532

[2] Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, et al. 2023. ANPL: towards natural programming with interactive decomposition. *Advances in Neural Information Processing Systems* 36 (2023), 69404–69440.

[3] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. 2023. LLM-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904* (2023). doi:10.48550/arXiv.2311.14904

[4] Yuling Shi, Songsong Wang, Chengcheng Wan, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215* (2024). doi:10.48550/arXiv.2410.01215

[5] Stack Overflow. 2024. Stack Overflow Developer Survey 2024. https://survey.stackoverflow.co/2024/

[6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

[7] Jiaxin Wen, Ruiqi Zhong, Pei Ke, Zhihong Shao, Hongning Wang, and Minlie Huang. 2024. Learning task decomposition to assist humans in competitive programming. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 11700–11723. doi:10.18653/v1/2024.acl-long.629

[8] Shangzi Xue, Zhenya Huang, Jiayu Liu, Xin Lin, Yuting Ning, Binbin Jin, Xin Li, and Qi Liu. 2024. Decompose, analyze and rethink: Solving intricate problems with human-like reasoning cycle. *Advances in Neural Information Processing Systems* 37 (2024), 357–385.

[9] Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. 2023. Parsel: Algorithmic Reasoning with Language Models by

Composing Decompositions. *Advances in Neural Information Processing Systems* 36 (2023), 31466–31523.

[10] Li Zhong and Zilong Wang. 2024. Can LLM replace Stack Overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 21841–21849. doi:10.1609/aaai.v38i19.30185

Received 2025-07-07; accepted 2025-08-08