

Lost in Code Generation: Reimagining the Role of Software Models in AI-driven Software Engineering

Jürgen Cito
TU Wien
Vienna, Austria

Dominik Bork
TU Wien
Vienna, Austria

Abstract

Generative AI enables rapid “vibe coding,” where natural language prompts yield working software systems. While this lowers barriers to software creation, it also collapses the boundary between prototypes and engineered software, leading to fragile systems that lack robustness, security, and maintainability. We argue that this shift motivates a reimagining of software models. Rather than serving only as upfront blueprints, models can be recovered post-hoc from AI-generated code to restore comprehension, expose risks, and guide refinement. In this role, models serve as mediators between human intent, AI generation, and long-term system evolution, providing a path toward sustainable AI-driven software engineering.

1 Vibe Coding: A Tragedy in Three Acts

In early 2024, a non-technical creator, Leonel Acevedo (@leojr94_ on X), documented on X how he had built and launched a Software-as-a-Service application using nothing but AI coding assistants.¹ In a matter of days, he had moved from idea to deployed system, while not writing a single line of code himself. The practice he described as *vibe coding* consisted of expressing functionality in natural language, iteratively pasting instructions into tools such as Cursor,² and accepting the generated outputs as the foundation of his product. Remarkably, this process was not just an academic experiment. Acevedo acquired paying customers and operated a real online business.

Act I: The Good. This account captures much of the optimism surrounding generative AI in software engineering. It highlights a world where barriers to entry are radically lowered, where anyone can instantiate working software from a loosely expressed idea, and where entrepreneurial imagination can translate into deployed systems without requiring years of technical training. Or, as Matt Welsh in a recent viewpoint provocatively asks, do we face: “*The End of Programming*”? [40]. In this sense, vibe coding resonates strongly with decades of aspirations for model-driven engineering, low-code, and citizen developers [8, 35]. Generative AI seems to fulfill these promises in a way that previous paradigms never fully achieved: instead of painstakingly learning abstractions, users can directly describe intent and obtain running systems. For the first time, the longstanding vision of natural language-oriented programming and democratizing software creation [4, 37] appears tangible.

Act II: The Bad. However, the story does not end with celebration. As Acevedo’s user base expanded, familiar difficulties emerged. The application started to suffer from performance bottlenecks, scalability issues, and brittleness under load. What initially appeared as a production-ready system became a prototype in disguise. This is

an old tale: prototyping has long been valued for its ability to surface feedback early, but it has equally long been criticized for seducing stakeholders into mistaking appearance for substance [6, 17, 21, 33]. The system *looked* complete, but its underlying quality attributes had not been engineered.

We see intriguing parallels to software prototyping [3, 9, 16]. Prototypes were always meant to be disposable, a means of validating requirements and exploring ideas. However, once they began to resemble finished products, the temptation to deploy them as such became overwhelming. The same dynamic is repeating itself now, except that generative AI enables much higher-fidelity prototypes. These are not mock-ups or half-coded shells; rather, they are fully functional applications. Because they run, connect to APIs, and offer complete features, users assume that they must also meet non-functional requirements. Unlike experienced software engineers, who typically know which aspects are missing in a prototype, non-technical vibe coders are rarely aware of these gaps.

Act III: The Ugly. The narrative took a further turn when Acevedo began reporting serious security problems. Malicious actors discovered ways to bypass subscription checks, exploit exposed API keys, and inject arbitrary data into the database. The product that had seemed a triumph of accessibility and speed now appeared dangerously fragile. In his own words: “*guys, I’m under attack, ever since I started to share how I built my SaaS using Cursor, random things are happening, maxed out usage on API keys, people bypassing the subscription, creating random stuff on db.*”³ These reports illustrate the darker side of generative AI for code generation: superficially functional systems, once exposed to adversarial environments, can collapse catastrophically.

The episode demonstrates that the issues are not accidental missteps of a single creator but structural consequences of how generative AI currently mediates software creation. When code is generated directly from natural language, no explicit intermediate artifacts are produced that make the system’s structure, requirements,⁴ assumptions, or limitations visible. What results is software that may work superficially but is essentially a black box, even to its creator.

Beyond the Lone Vibe Coder. While this narrative is evocative, we contend that it reflects a broader pattern, not only among non-technical solo creators but even within engineering teams of varying experience. Junior developers often concentrate on achieving functional correctness, paying limited attention to architecture, maintainability, or non-functional concerns [14, 25]. Even in large-scale systems, experienced engineers often find themselves without

¹https://x.com/leojr94_/status/1900767509621674109

²<https://cursor.com/>

³https://x.com/leojr94_/status/1901560276488511759

⁴The prompting history could be seen as an implicit account of requirements, but is often not visible, and even if it were, reading and comprehending this history is time-consuming and prone to error

sufficient time, perspective, or appropriate abstractions to holistically reason about generated or scaffolded code [10, 22]. In team settings, the challenge is compounded: one must not only make sense of one’s own AI-generated code but also understand the “vibe-coded” fragments produced by colleagues, without always knowing their prompting intent and the conversation history.

Empirical reports already hint at such risks. In one dramatic case, an autonomous AI agent in Replit deleted a production database despite explicit instructions to freeze code changes, even fabricating fake records and hiding its actions [1]. More broadly, reports from EclipseSource, a software engineering company that works with enterprise-level codebases, describe the recurring difficulties in integrating AI into established development practice. Teams often lack predefined workflows to incorporate AI agents, generic prompts misalign with legacy or domain-specific constraints, and insufficient training or alignment hampers effective adoption [13]. Finally, in an interview study of engineers who participated in AI-assisted code review, the participants described that LLM suggestions sometimes imposed a higher cognitive load due to verbosity, lacked explanatory depth, and were adopted only with skepticism when context was missing [2]. Together, these accounts suggest that the opacity and disconnect introduced by direct generation challenge not just *novices* or users with non-technical backgrounds but also professional software engineering teams.

Reimagining the Role of Software Models. The good, the bad, and the ugly of vibe coding illustrate both the excitement and the risks of generative AI. On the one hand, it democratizes software creation and lowers barriers to entry. On the other hand, it revives the old gap between rapid prototypes and engineering-grade systems, while also exposing creators to severe failures once these systems face real-world conditions [29, 30].

We argue that these developments should motivate a reimagining of software models. Traditionally, conceptual models, such as entity-relationship diagrams, UML class and sequence diagrams, or state machines, provided a blueprint for developers and a shared language for stakeholders. With generative AI, however, this sequencing is inverted: functionality is described in natural language, code is generated, and deployed systems emerge without explicit models. The consequence is that developers and users alike often lack a coherent understanding of what the system actually does.

Rather than being discarded, models can assume a new role as post-hoc and bidirectional instruments. By reverse-engineering abstractions from AI-generated software systems, we can recover an intelligible view of structure and behavior [18], identify risks, and expose limitations. At the same time, such models need not remain static. They can feed back into the generative loop, constraining and guiding subsequent code generation. In this sense, models become key mediators between human intent, AI generation, and the long-term evolution of software systems.

2 From Prototyping to Vibe Coding

The history of software engineering has repeatedly confronted the tension between speed and rigor [5, 22]. Prototyping emerged as a practice precisely to address this tension: it allowed developers and stakeholders to explore requirements, validate assumptions, and test user interactions long before a full system was engineered. The

core value of prototyping lies in its ability to make ideas tangible at an early stage, thus supporting dialogue and reflection. But the very success of prototypes has always carried risks. Once a prototype acquired a polished interface, it became difficult for stakeholders to resist the temptation to treat it as a finished product [36, 41]. The gap between appearance and substance, i.e., between what was demonstrated and what was actually engineered, has been a recurring theme in software practice for decades.

From Illusion to Reality. This historical experience is highly relevant to the phenomenon of vibe coding. Generative AI enables the production of software artifacts that go far beyond the scope of traditional prototypes. They are not sketches, partial simulations, or interactive wireframes. They are executable systems that can be deployed to the cloud, connect to payment providers, and serve real users. Their polish, responsiveness, and functionality make them appear indistinguishable from production-grade systems. The illusion is no longer merely visual but functional.

However, beneath the surface, the same risks persist. The generated code often lacks considerations of robustness, efficiency, or security [31]. While the system runs, it may do so only under narrowly defined conditions. Edge cases, concurrency issues, and adversarial interactions are rarely accounted for. The result is that creators are misled by the apparent completeness of the artifact, believing that they have built a viable product when they have in fact produced an elaborate prototype.

Shifting Knowledge Boundaries. There is, however, an important difference between earlier experiences with prototyping and the current wave of AI-assisted development. Traditional prototypes were created by engineers who generally understood which corners were being cut. They knew that data validation had been deferred, that security concerns had not been addressed, or that performance had not been optimized. This knowledge acted as a safeguard, even if the temptation to deploy prototypes prematurely remained.

In vibe coding, by contrast, many creators are non-experts. They may have entrepreneurial vision or domain expertise, but lack the technical background to appreciate which aspects of the system are fragile or incomplete. For them, the generated system is a black box: if it runs, it is assumed to be correct; if it looks professional, it is assumed to be robust. The epistemic asymmetry between traditional prototyping and AI-driven prototyping is thus profound. What was once a conscious trade-off made by engineers becomes an unconscious risk borne by non-engineers.

Acceleration and Amplification. Generative AI does more than reproduce the pitfalls of prototyping; it amplifies them. The speed with which a prototype can be turned into a running system is unprecedented. Deployment pipelines, cloud hosting platforms, and API ecosystems enable reaching users within days or even hours. This acceleration collapses the temporal buffer that once separated prototyping from production. What used to be a deliberate decision to “harden” a prototype is now bypassed entirely, as systems transition from generation to deployment in a single continuous step.

The consequences of this shift are significant. The boundary between prototypes and products is dissolving, not just in perception, but in practice as well. Systems generated by AI are launched into real-world contexts without ever undergoing the processes of design review, architectural planning, or quality assurance that have historically characterized software engineering. The familiar dangers of prototyping are thus not only repeated, but intensified, with risks manifesting earlier, more suddenly, and more severely.

A Structural Predicament. Seen in this light, vibe coding is not simply a new name for prototyping. It represents a structural transformation of how software is conceived, produced, and deployed. The traditional cycle, requirements, design, implementation, and testing, assumes that models and abstractions exist before code. Vibe coding inverts this order: code comes first, models (if they exist at all) follow later. The result is a new epistemic landscape in which understanding lags behind generation. Developers, users, and even the creators themselves struggle to make sense of what has been produced [32].

This structural inversion creates the conditions for the kinds of failure described earlier: performance bottlenecks, brittleness under load, and security breaches. Without explicit abstractions to mediate between intention and implementation, the system remains opaque. The problem is not simply that vibe coding is “bad prototyping.” It is the very act of generation without understanding through modeling that displaces the foundation upon which sustainable engineering practices have traditionally rested [11, 12].

3 Reimagining the Role of Models

From Prequels to Post-Hoc Artifacts. Conceptual models have traditionally been employed as prequels to programming. They served as blueprints: abstractions that guided implementation and mediated communication among stakeholders [28]. These models did not merely describe software; they prescribed how it should be built. Their epistemic role was anticipatory: they structured thinking and formalized communication in advance of coding [19].

Generative AI inverts this sequencing. The conversational interface with LLMs enables direct translation from natural language to executable code, thereby bypassing the need for modeling altogether. Functionality is no longer abstracted and reasoned about prior to implementation; it is expressed, generated, and deployed in a single step. Instead of models guiding construction, they are absent altogether, leaving only code behind. This creates an epistemic void: what exists runs, but what it actually does remains opaque.

Models as Instruments of Understanding. We argue that this inversion invites a rethinking of the role of models. Rather than being abandoned as relics of a pre-generative era, models can be repositioned as post-hoc artifacts. By applying reverse engineering techniques, it is possible to recover structural and behavioral abstractions from AI-generated systems. These models then serve as tools that foster understanding. They reveal architectural structures that may otherwise be hidden, illuminate data flows, and make visible the assumptions embedded in the generated code. For developers, this recovery process provides a map of the terrain they must navigate. For non-technical creators, it offers a conceptual scaffold that explains what their systems actually do.

Bidirectionality and Co-Evolution. Once recovered, models do not need to remain static documentation. Their real potential lies in their potential to become bidirectional instruments. Modifications at the model level, such as refining relationships, restructuring behaviors, or specifying constraints, can be translated into new prompts that guide subsequent rounds of code generation. The process is no longer one-way, from model to code, but cyclical, with code and models co-evolving over time. This bidirectionality restores the alignment between human intent and machine execution that is otherwise lost in direct natural language programming.

Synergy with Generative AI. Models also offer a way to foster generative AI itself. While LLMs excel at pattern recognition and code synthesis, they struggle to maintain consistent global structure or reasoning across abstraction levels [18]. Models can provide structured, machine-readable representations that complement the probabilistic tendencies of LLMs. For instance, a recovered data model can help prevent inconsistencies in schema usage; a behavioral model can constrain the space of valid interactions; an architectural model can serve as an anchor for optimization or verification tasks; a domain model can formally prescribe the context in which the software is to be used. In this sense, models are not merely aids for humans but also scaffolds for AI systems, ontologically grounding their outputs in explicit abstractions [19].

Toward a New Epistemic Role. Reimagined in this way, models are no longer prequels to code but mediators of intent, comprehension, refinement, and evolution. They help bridge the gap between rapid generation and sustainable engineering by restoring a level of transparency that pure vibe coding lacks. The significance of this shift is not only technical but epistemic: it changes how we perceive software. Instead of understanding emerging from planned abstractions that precede code, understanding is reconstructed after the fact, through the recovery and refinement of models. This repositioning does not return us to traditional model-driven engineering; rather, it opens a new trajectory in which models play a different, yet equally central, role in the software lifecycle.

Beyond Passive Generation. If we accept that vibe coding produces opaque prototypes and that models can be repurposed to foster understanding and refinement, the natural question becomes: what comes next? Currently, LLMs are primarily used as passive generators [30]. The burden of validation, correction, and evolution remains, to a large extent, a human task. This passivity sharply limits the potential of generative AI in software engineering.

The Promise of Agency. Recent discussions in the community point toward a paradigm of *agentic software engineering* [20]. The idea is that AI systems need not remain mere assistants that produce artifacts on demand. Instead, they can be configured as autonomous or semi-autonomous agents that monitor, test, optimize, and adapt software. Such agents would not only respond to prompts but also proactively engage in the system’s lifecycle. They could identify anomalies, suggest improvements, and enforce constraints. Importantly, they would act in collaboration with human developers, forming mixed teams of humans and AI systems.

Models as Mediators of Agency. For such agentic workflows to succeed, however, a medium of coordination is required. Raw

code is too fine-grained, too brittle, and too inconsistent for multiple agents (human or artificial) to collaborate effectively [39]. Models provide precisely the abstractions needed to mediate this collaboration by addressing a crucial need, semantic understanding [18, 19]. Structural models can serve as shared representations of architecture, ensuring that different agents reason about the same components and interfaces. Behavioral models can delineate expected interactions, against which agents can check conformity. Data models can anchor consistency across schema evolution and API usage. By operating at this higher level of abstraction, agents can reason, plan, and coordinate their actions without getting lost in the details of generated code.

4 Future Plans: Implications, Challenges, and a Research Agenda

This vision of a reimagined role for software models in AI-driven software engineering presents several implications and challenges, forming a research agenda for the upcoming years.

Toward Sustainable AI-Driven Engineering. In our vision, vibe coding is not the endpoint, but the entry point of a longer trajectory, from a prototype being produced from natural language descriptions to model recovery, in which abstractions are extracted to render the system intelligible. The last crucial step is bidirectional refinement: models and code co-evolve, with modifications at one level informing changes at the other to transform vibe-coded prototypes into sustainable, evolving software systems. The goal is not to replace human developers, but to augment them with new forms of agency. The implications of this shift suggest that the future of generative AI in software engineering lies not in one-off code generation but in sustained, longitudinal engagement with software systems.

Models play a central role in a future where it becomes possible to envision a sustainable practice of AI-driven engineering, in which prototypes evolve and harden into robust systems through the combined efforts of human and AI agents. *A key research question is how models can be designed and operationalized so that they not only document AI-generated systems but actively support their continuous monitoring, repair, and evolution by both human developers and autonomous AI agents.*

The Challenge of Fidelity. If models are to serve as mediators, their fidelity to the underlying system is paramount (cf. [27]). Reverse engineering software models from AI-generated code is not a trivial matter. Code produced by LLMs is often inconsistent, redundant, or ad hoc, lacking the disciplined structure of manually engineered systems [23, 38]. Extracting clean architectural or behavioral abstractions from such artifacts requires sophisticated techniques that can tolerate noise and ambiguity. Fidelity here is not only a technical challenge but also an epistemic one: models must faithfully capture what the system does without falsely suggesting a level of order or intentionality that does not really exist. *A key research question is therefore how to balance accuracy with interpretability when reconstructing models from generated systems.*

Managing Levels of Abstraction. Another challenge lies in choosing the appropriate modeling notations and levels of abstraction (cf. [15]). Traditional software modeling offered a wide spectrum, from high-level conceptual schemas to low-level design diagrams. In the context of generative AI, the question is not simply which abstractions are technically possible, but which are useful for guiding both human and AI agents. A level of abstraction that is too high risks making the model a vague generality that cannot effectively constrain code generation. Too low a level, and it becomes indistinguishable from source code, losing its explanatory power. *A key research question is whether current modeling notations can strike this balance for models to function as practical mediators in AI-assisted software engineering or whether new notations need to be developed.*

Trust, Verification, and Risk Awareness. One of the promises of reintroducing models is that they might surface risks that are invisible in raw code. Security vulnerabilities, performance bottlenecks, and architectural inconsistencies often manifest more clearly at the model level than in thousands of lines of generated code [24, 26]. But this potential creates a parallel responsibility: models must be trustworthy, and their analyses must be reliable. Incorrect or misleading models could induce a false sense of security, leaving creators exposed to the very dangers models were meant to prevent. *Verification of recovered models and methods for measuring model uncertainty or incompleteness are thus crucial research areas.*

Accessibility for Non-Experts. Perhaps the most pressing challenge is accessibility. The appeal of vibe coding lies precisely in its ability to empower non-technical users to produce functioning systems. If models are to play a central role in this ecosystem, they must be designed with these users in mind. Traditional modeling languages are often criticized as too formal, too abstract, or too cumbersome for non-experts [7, 34]. Simply reintroducing them will not suffice. *Instead, new forms of representation may be needed: lightweight notations, visual interfaces, or natural-language explanations that convey the essence of a system without demanding expertise in modeling theory.* Achieving this balance is crucial if the democratizing promise of generative AI is to be preserved while also mitigating its risks.

5 Conclusion

The story of vibe coding illustrates both the promise and peril of generative AI in software engineering. It democratizes access to software creation, yet simultaneously collapses the boundary between prototype and product, exposing creators to risks they cannot see. We have argued that software models can be reimagined to address this predicament. By recovering, analyzing, and refining models post-hoc, we can restore a measure of transparency and control to AI-generated systems. This is not the only path forward, nor do we claim it as a panacea. But it offers one promising avenue: to treat models not as relics of upfront design but as dynamic, bidirectional instruments of comprehension and evolution. If vibe coding is the new prototyping, then models may serve as the compass that prevents us from getting truly lost in code generation.

References

- [1] Aminu Abdullahi. 2025. AI Agent Wipes Production Database, Then Lies About It. Public report via social media, summarized in industry blogs. <https://www.eweek.com/news/replit-ai-coding-assistant-failure/> Replit incident deleting production database despite freeze instructions.
- [2] Adam Alami and Neil Ernst. 2025. Human and Machine: How Software Engineers Perceive and Engage with AI-Assisted Code Reviews Compared to Their Peers. In *2025 IEEE/ACM 18th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*. 63–74. doi:10.1109/CHASE66643.2025.00016
- [3] Jonathan Arnowitz, Michael Arent, and Nevin Berger. 2010. *Effective prototyping for software makers*. Elsevier.
- [4] Amin Beheshti. 2024. Natural Language-Oriented Programming (NLOP): Towards Democratizing Software Creation. In *IEEE International Conference on Software Services Engineering, SSE 2024, Shenzhen, China, July 7-13, 2024*. IEEE, 258–267. doi:10.1109/SSE62657.2024.00047
- [5] Patrik Berander, Lars-Ola Damm, Jeanette Eriksson, Tony Gorschek, Kennet Henningsson, Per Jönsson, Simon Kågström, Drazen Milicic, Frans Mårtensson, Kari Rönkkö, et al. 2005. Software quality attributes and trade-offs. *Blekinge Institute of Technology* 97, 98 (2005), 19.
- [6] Elizabeth Bjarnason, Franz Lang, and Alexander Mjöberg. 2023. An empirically based model of software prototyping: a mapping study and a multi-case study. *Empirical Software Engineering* 28, 5 (2023), 115.
- [7] Dominik Bork, Stefan Klikovits, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2025. Inclusive Model-Driven Engineering for Accessible Software. In *MODELS 2025 NIER - Int. Conf. on Model Driven Engineering Languages and Systems*. <https://model-engineering.info/publications/papers/25-MODELS-NIER.pdf>
- [8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice*. Morgan & Claypool Publishers.
- [9] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighoven. 1990. What is prototyping? *Information Technology & People* 6, 2/3 (1990), 89–95.
- [10] Janet E. Burge and David C. Brown. 2003. Rationale Support for Maintenance of Large Scale Systems. In *Workshop on Evolution of Large Scale Industrial Software Applications (ELISA 2003)*.
- [11] Charles E Dickerson and Dimitri Mavris. 2013. A brief history of models and model based systems engineering and the case for relational orientation. *IEEE Systems Journal* 7, 4 (2013), 581–592.
- [12] Claudia Eckert and Rafaela Hillerbrand. 2022. Models in engineering design as decision-making aids. *Engineering Studies* 14, 2 (2022), 134–157.
- [13] EclipseSource. 2025. Why AI Coding Still Fails in Enterprise Teams. Blog post. <https://eclipsesource.com/blogs/2025/06/11/why-ai-coding-fails-in-enterprises/>
- [14] Matteo Eposito, Andrea Janes, Terhi Kilamo, and Valentina Lenarduzzi. 2025. Early career developers' perceptions of code understandability: a study of complexity metrics. *IEEE Access* (2025).
- [15] Hans-Georg Fill, Jordi Cabot, Wolfgang Maass, and Marten van Sinderen. 2024. AI-Driven Software Engineering - The Role of Conceptual Modeling. *Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model.* 19 (2024). doi:10.18417/EMISA.19.1
- [16] Christiane Floyd. 1984. A systematic look at prototyping. In *Approaches to prototyping*. Springer, 1–18.
- [17] Elizabeth Gerber and Maureen Carroll. 2012. The psychological experience of prototyping. *Design studies* 33, 1 (2012), 64–84.
- [18] Alex Gu, Naman Jain, Wen-Ding Li, Manish Shetty, Yijia Shao, Ziyang Li, Diyi Yang, Kevin Ellis, Koushik Sen, and Armando Solar-Lezama. 2025. Challenges and Paths Towards AI for Software Engineering. *CoRR* abs/2503.22625 (2025). arXiv:2503.22625 doi:10.48550/ARXIV.2503.22625
- [19] Giancarlo Guizzardi, Oscar Pastor, and Veda C. Storey. 2023. Thinking Fast and Slow in Software Engineering. *IEEE Softw.* 40, 6 (2023), 139–142. doi:10.1109/MS.2023.3306132
- [20] Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. 2025. Agentic Software Engineering: Foundational Pillars and a Research Roadmap. arXiv:2509.06216 [cs.SE] <https://arxiv.org/abs/2509.06216>
- [21] Aleksander Jarzbowicz and Pawel Weichbroth. 2021. A qualitative study on non-functional requirements in agile software development. *IEEE Access* 9 (2021), 40458–40475.
- [22] Miikka Kuuttila, Mika Mäntylä, Umar Farooq, and Maëlick Claes. 2020. Time pressure in software engineering: A systematic review. *Information and Software Technology* 121 (2020), 106257. doi:10.1016/j.infsof.2020.106257
- [23] Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, et al. 2025. Code Copycat Conundrum: Demystifying Repetition in LLM-based Code Generation. *arXiv preprint arXiv:2504.12608* (2025).
- [24] Mohammed Misbhaudhin and Mohammad R. Alshayeb. 2015. UML model refactoring: a systematic literature review. *Empir. Softw. Eng.* 20, 1 (2015), 206–251. doi:10.1007/S10664-013-9283-7
- [25] Simona Motogna, Diana Cristea, Diana-Florina Sotropa, and Arthur-Jozsef Molnar. 2024. Uncovering Bad Practices in Junior Developer Projects Using Static Analysis and Formal Concept Analysis.. In *ENASE*. 752–759.
- [26] Haris Mumtaz, Mohammad R. Alshayeb, Sajjad Mahmood, and Mahmood Niaz. 2019. A survey on UML model smells detection techniques for software refactoring. *J. Softw. Evol. Process.* 31, 3 (2019). doi:10.1002/SMR.2154
- [27] Paula Muñoz, Manuel Wimmer, Javier Troya, and Antonio Vallecillo. 2024. Measuring the Fidelity of a Physical and a Digital Twin Using Trace Alignments. *IEEE Trans. Software Eng.* 50, 12 (2024), 3122–3145. doi:10.1109/TSE.2024.3462978
- [28] John Mylopoulos. 1992. Conceptual modelling and Telos. In *Conceptual modelling, databases, and CASE: An integrated view of information system development*. Chichester, UK: John Wiley & Sons, 49–68.
- [29] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q. Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)read Each Other. In *Proceedings of the CHI Conference on Human Factors in Computing Systems, CHI 2024, Honolulu, HI, USA, May 11-16, 2024*, Florian 'Floyd' Mueller, Penny Kyburz, Julie R. Williamson, Corina Sas, Max L. Wilson, Phoebe O. Toups Dugas, and Irina Shklovski (Eds.). ACM, 651:1–651:26. doi:10.1145/3613904.3642706
- [30] Anh Nguyen-Duc, Beatriz Cabrero Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, Mika Saari, Zheyang Zhang, Huy Le, Tho Quan, and Pekka Abrahamsson. 2025. Generative artificial intelligence for software engineering—A research agenda. *Software: Practice and Experience* (2025). doi:10.1002/spe.70005
- [31] Marten Oltrogge, Erik Derr, Christian Stransky, Yasemin Acar, Sascha Fahl, Christian Rossow, Giancarlo Pellegrino, Sven Bugiel, and Michael Backes. 2018. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 634–647. doi:10.1109/SP.2018.00005
- [32] James Prather, Brent N. Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S. Randerianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1, ICER 2024, Melbourne, VIC, Australia, August 13-15, 2024*, Paul Denny, Leo Porter, Margaret Hamilton, and Briana B. Morrison (Eds.). ACM, 469–486. doi:10.1145/3632620.3671116
- [33] Laura Rehberg and Alexander Brem. 2025. Bridging the gap: Linking prototyping and technology readiness levels for integrative product development. *Creativity and Innovation Management* 34, 1 (2025), 237–252.
- [34] Aylin Sarioglu, Haydar Metin, and Dominik Bork. 2025. Accessibility in conceptual modeling - A systematic literature review, a keyboard-only UML modeling tool, and a research roadmap. *Data Knowl. Eng.* 158 (2025), 102423. doi:10.1016/J.DATAK.2025.102423
- [35] Phil Simon. 2022. *Low-Code/No-Code: Citizen Developers and the Surprising Future of Business Applications*. Racket Publishing.
- [36] Paul Swail. 2022. When Prototype Becomes Production. <https://serverlessfirst.com/emails/when-prototype-becomes-production>. Accessed: 2025-09-29.
- [37] Eric Von Hippel. 2006. *Democratizing innovation*. the MIT Press.
- [38] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, Mingzhi Mao, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 690–712.
- [39] Yun Wang and Yan Lu. 2025. Interaction, Process, Infrastructure: A Unified Architecture for Human-Agent Collaboration. *arXiv preprint arXiv:2506.11718* (2025).
- [40] Matt Welsh. 2023. The End of Programming. *Commun. ACM* 66, 1 (2023), 34–35. doi:10.1145/3570220
- [41] Shai Yallin. 2021. Permanent Prototypes. <https://www.shaiyallin.com/post/permanent-prototypes>. Accessed: 2025-09-29.