

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

Jürgen Cito

Software Engineering Group, TU Wien  
Vienna, Austria  
[juergen.cito@tuwien.ac.at](mailto:juergen.cito@tuwien.ac.at)

Dominik Bork

Business Informatics Group, TU Wien  
Vienna, Austria  
[dominik.bork@tuwien.ac.at](mailto:dominik.bork@tuwien.ac.at)

## Abstract

Generative AI enables rapid “vibe coding” and agentic software engineering, where natural language prompts yield working software systems. This lowers barriers to software creation, but it also collapses the boundary between prototypes and engineered software. Systems that appear complete may lack robustness, security, and maintainability. We argue that this shift motivates a renewed role for software models. Rather than serving only as upfront blueprints, models can be recovered from AI-generated systems, used to restore comprehension, and refined to guide subsequent evolution. We distinguish an agentic loop, in which recovered models support automated checking and repair, from a human reflective loop, in which models expose assumptions for inspection and revision. We identify candidate model types and illustrate how recovered models can expose implicit assumptions as possible constraints for review and validation. This paper positions software models as a bridge between current software engineering practice and model-based reasoning in AI-driven development.

## CCS Concepts

• **Software and its engineering**; • **Computing methodologies**  
→ **Artificial intelligence**; • **Human-centered computing**;

## Keywords

Vibe coding, generative AI, agentic software engineering, software models, model-driven engineering, software comprehension

### ACM Reference Format:

Jürgen Cito and Dominik Bork. 2026. Lost in Code Generation: Reimagining the Role of Software Models in AI-driven Software Engineering. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS 2026)*, October 04–09, 2026, Málaga, Spain. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3822455.3838778>

## 1 Vibe Coding in Three Acts

Vibe coding and agentic software engineering are no longer fringe curiosities. Natural-language programming environments, AI coding assistants, and increasingly autonomous development agents are now used to create prototypes, internal tools, production services, and changes to existing codebases. This shift makes the original vibe-coding story less exceptional than it first appeared.

Alongside this optimism, recent public accounts collect failures across several platforms, including exposed API keys, inverted access control, authentication bypasses, remote code execution, and agentic deletion of production data.<sup>1</sup> The specific cases differ, but

<sup>1</sup>See the industry overview at <https://getautonoma.com/blog/vibe-coding-failures>. We use this as a motivating collection of incidents rather than as a primary empirical study.

they point to the same structural problem. *AI-generated software can be demo-complete and locally plausible while remaining engineering-incomplete, with assumptions about security, data protection, runtime boundaries, and evolution left implicit.*

One early and memorable example was documented by Leonel Acevedo (@leojr94\_ on X), a non-technical creator who built and launched a Software-as-a-Service application using AI coding assistants.<sup>2</sup> In a matter of days, he moved from idea to deployed system without 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,<sup>3</sup> 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*”? [43] Vibe coding resonates strongly with decades of aspirations for model-driven engineering, low-code, and citizen development [8, 36]. Generative AI seems to fulfill parts of 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 [3, 40].

**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. The system *looked* complete, but its underlying quality attributes had not been engineered. Section 2 develops this parallel in more detail.

**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,*

<sup>2</sup>[https://x.com/leojr94\\_/status/1900767509621674109](https://x.com/leojr94_/status/1900767509621674109). We use this public account as an illustrative anecdote, not as empirical evidence about the prevalence or reliability of vibe coding.

<sup>3</sup><https://cursor.com/>

*random things are happening, maxed out usage on API keys, people bypassing the subscription, creating random stuff on db.*<sup>4</sup> These reports illustrate the darker side of generative AI for code generation. Superficially functional systems, once exposed to adversarial environments, can collapse catastrophically.

The point is not that one creator made avoidable mistakes. The broader issue is 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,<sup>5</sup> 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.** The early vibe-coding narrative centered on solo non-experts, but the same opacity now matters in professional settings. Junior developers often concentrate on achieving functional correctness, paying limited attention to architecture, maintainability, or non-functional concerns [14, 26]. Even in large-scale systems, experienced engineers often lack sufficient time, perspective, or appropriate abstractions to holistically reason about generated or scaffolded code [9, 23]. In team settings, the challenge is compounded. One must not only make sense of one's own AI-generated code but also understand the fragments generated by colleagues or agents, without always knowing their prompting intent or conversation history.

Public incidents, industry reports, and early empirical studies point to related 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]. Reports from EclipseSource describe 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, participants described that LLM suggestions sometimes imposed 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 introduced by direct generation challenges novices and professionals alike.

**Scope of the Claim.** We do not claim that vibe coding has replaced disciplined engineering in large, regulated, or safety-critical systems. Established standards, reviews, architecture work, and coordination practices remain essential in such settings. Our target is the growing middle ground in which AI-generated prototypes become consequential, agents produce changes in existing repositories, and teams inherit generated fragments that must be understood, validated, and hardened.

**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 [30, 31]. They also motivate the central

claim of this paper. Software models should not be treated only as upfront design artifacts. In AI-driven software engineering, they can mediate between human intent, generated code, validation, and long-term evolution. We develop this claim through a prototyping analogy, a concrete model-mediated feedback loop, candidate model types, and research tasks that make the proposal actionable.

## 2 From Prototyping to Vibe Coding

The history of software engineering has repeatedly confronted the tension between speed and rigor [5, 23]. 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 [39, 44]. The gap between appearance and substance, 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 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.

Beneath the surface, familiar risks persist. The generated code often lacks considerations of robustness, efficiency, or security [32]. Edge cases, concurrency issues, and adversarial interactions are rarely accounted for. The illusion is no longer merely visual but also functional, making the gap between prototype and product harder to recognize.

**Shifting Knowledge Boundaries.** However, there is an important difference between earlier prototyping experiences 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 served as a safeguard, even as the temptation to deploy prototypes prematurely persisted.

In vibe coding, by contrast, many creators are non-experts. They may have entrepreneurial vision or domain expertise, but lack the technical background to understand 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. Deployment pipelines, cloud hosting platforms, and API ecosystems

<sup>4</sup>[https://x.com/leojr94\\_/status/1901560276488511759](https://x.com/leojr94_/status/1901560276488511759)

<sup>5</sup>The 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.

enable users to be reached 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 can be bypassed as systems move from generation to deployment in a single continuous step.

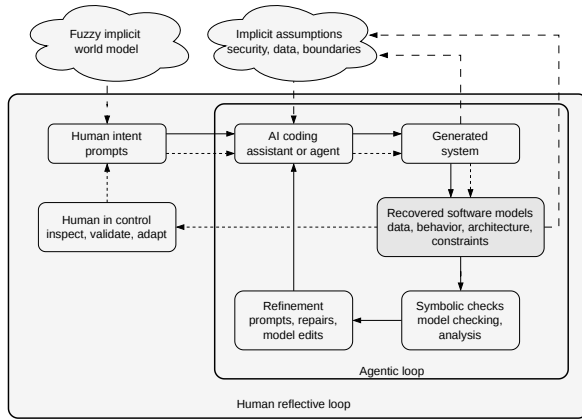


**Figure 1: Software models as mediators in an agentic loop between prompt-driven generation, automated checking, refinement, and human reflection.**

**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 of requirements, design, implementation, and testing assumes that models and abstractions exist before code. Vibe coding inverts this order. Code comes first, models follow later if they exist at all. 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 [34].

This structural inversion creates the conditions for the kinds of failure described earlier. Without explicit abstractions to mediate between intention and implementation, the system remains opaque [4]. The problem is not simply that vibe coding is bad prototyping. It is the act of generation without understanding through modeling that displaces the foundation upon which robust, maintainable engineering practices have traditionally rested [11, 12].

### 3 Software Models as Mediators

**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 [29]. 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 [18].

As argued above, generative AI often inverts this sequencing. Functionality is expressed in natural language, generated as code, and only later subjected to engineering scrutiny. The modeling

question therefore changes. Instead of asking only how models can guide implementation, we must also ask how models can recover and stabilize understanding after implementation.

**Model-Mediated Feedback.** We propose model-mediated feedback loops for AI-generated software. They treat generated code as an intermediate step in engineering, not as the final artifact. Fig. 1 shows how these operations support an *automatic agentic loop* and a *human reflective loop*.

**Two Complementary Loops.** In the *agentic loop*, generated code is not merely handed back to an AI assistant for another edit. It is abstracted into expected models of data, behavior, architecture, runtime context, or constraints. These models provide a symbolic handle on the generated system. Model checking, program analysis, tests, or policy checks can then compare the expected model with the implementation and produce concrete feedback. That feedback can become a repair prompt, a generated regression test, a static check, a model edit, or an instruction that constrains later agent actions. Recovered model elements remain linked to their supporting code, tests, or traces and distinguish observed behavior from inferred or asserted constraints. Once those constraints are validated by a human or authoritative policy, violations can produce evidence-linked counterexamples, regression tests, or repair prompts; recovery and checking then repeat after repair.

The *human reflective loop* uses the same recovered models in different ways. Here, the models are not primarily inputs to an automatic repair step. They represent assumptions that were implicit in the human’s world model, the prompt, the agent’s interpretation, and the generated code. By inspecting the recovered model, the human can notice that a concept is missing, a boundary is wrong, or a security assumption was never made explicit. The resulting feedback goes back into the human’s intent and into the next prompt or model edit. This makes the model a mediating artifact between the human in the control seat and the agentic generation loop.

**Model Types and Workflow Fit.** Table 1 makes the loop more concrete. The goal is not to recover every possible model or to impose heavyweight modeling before a user can continue working. Instead, the loop should appear at natural pauses in a vibe coding workflow. It can run before deployment, after an agentic change, when tests fail, when a prototype gains users, or during review. At these moments, recovered models provide a compact representation that is easier to inspect than a large prompting history and the correspondingly generated codebase. The value of these models is strongest when they remain linked to code, tests, traces, and prompts, so that users can move from an abstract concern to the artifact that supports or violates it. This does not make model recovery trivial. Reverse engineering design intent from code remains incomplete and uncertain, especially when generated code contains ad hoc structure or when the rationale behind prior changes is unavailable [9, 24, 41]. The challenge is therefore to recover useful partial models with explicit evidence and uncertainty rather than to promise perfect reconstruction. For humans working under time pressure or with verbose AI-generated feedback, such models should reduce the amount of code and prompting history that must be inspected before a judgment can be made [2, 23].

**Table 1: Candidate software models for post-hoc mediation in AI-generated systems.**

| Model type            | Recovery sources                                                     | Primary use                                                                                         | Example risk surfaced                                                          |
|-----------------------|----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| Domain and data model | Database schemas, ORM classes, API payloads, logs                    | Help creators, developers, and agents inspect core concepts, relationships, and business invariants | Missing relation between subscription state, usage records, and payment status |
| Behavioral model      | Routes, traces, tests, event logs, controller code                   | Check request ordering, protocols, and state changes                                                | External API call occurs before server-side authorization                      |
| Architectural model   | Imports, dependencies, service boundaries, deployment manifests      | Reveal component boundaries and isolate responsibilities                                            | Secrets or database access leak into client-facing code                        |
| Constraint model      | Assertions, policies, tests, prompts, inferred invariants            | Turn intent into checks, repair prompts, or generation constraints                                  | Paid access, rate limits, or data ownership rules are implicit                 |
| Runtime model         | Infrastructure configuration, environment variables, monitoring data | Connect generated code to operational assumptions                                                   | Missing rate limits, exposed keys, or unsafe deployment defaults               |

**A Small Hardening Scenario.** Consider a generated SaaS application with users, subscriptions, API keys, payments, usage records, and an external AI service. The application works in a demonstration. A user signs up, enters payment details, receives access, and sends requests to the external service. Later, attackers discover that subscription checks can be bypassed and that usage can be charged to exposed keys.

Although Table 1 presents a broader repertoire, a first realization can focus on the behavioral and constraint models. The recovered behavioral model records the request path from authentication and subscription checking to usage recording and the external API call. Each transition links to the route or controller code, test, or trace that supports it. It can thereby show that the external call occurs before authorization, that usage is not associated with the authenticated user, or that a check exists only in client-side code. These are only *observations* about the implementation. In this concrete case, whether paid access and ownership are required remains an inferred assumption until the creator or an authoritative policy confirms it.

The constraint model then turns these observations into an explicit candidate rule. Every external API call must be preceded by a server-side subscription check and must be associated with a usage record owned by the authenticated user. In the human reflective loop, the rule gives the creator a reviewable statement of what paid access means and where that assumption must hold. At this point, it remains a hypothesis about intent rather than a guarantee. Once the creator or an authoritative policy validates it, the rule can be encoded in tests or checks and enforced within their stated coverage. In the agentic loop, the validated rule is checked against the behavioral model. A violation produces a counterexample linked to the offending transition and source evidence; this can generate a regression test and a repair prompt that moves the check to the server route or corrects usage ownership. Recovery and checking then run again on the repaired code. The model is thus an artifact through which intent is proposed, reviewed, validated, and operationalized.

**Models as Coordination Artifacts.** The same mechanism also explains why models matter for agentic workflows [19]. Agents that monitor, test, repair, or optimize software need shared context. Raw code is too fine-grained and unstable to serve as the only coordination medium [42]. Models can provide structured, machine-readable representations that complement LLMs’ local

code synthesis with global structure [16]. Domain models anchor vocabulary, behavioral models constrain interactions, architectural models expose boundaries, and constraint models make invariants explicit. In this role, models are useful to humans and to AI systems alike.

**Toward a New Epistemic Role.** Reimagined in this way, models are no longer only prequels to code. They become mediators of intent, comprehension, refinement, and evolution. The resulting lifecycle does not return us to traditional model-driven engineering. It opens a trajectory in which models remain central, but their centrality comes from making generated systems understandable and governable after code already exists.

## 4 Future Plans

The proposed role of models raises several challenges for software engineering and modeling research. We frame them as future plans for software models rather than as an agenda only for model-driven engineering, because the problem sits between current AI-assisted programming practice and the long tradition of model-based reasoning. The central question is how models can become useful working artifacts in AI-driven development without requiring a return to heavy upfront modeling.

**Fidelity and Traceability.** Recovered models must be faithful enough to support decisions without implying more order than the generated system contains. AI-generated code is often inconsistent, redundant, or ad hoc [24, 41]. A recovered model may simplify this code in useful ways, but it can also hide defects by making an accidental structure look intentional. A concrete research task is to attach evidence to model elements, such as source locations, tests, logs, traces, prompts, and confidence markers. Fidelity measures can then distinguish observed facts from inferred intent and unsupported assumptions [28]. This limit is structural, not merely an engineering problem. Evidence that is absent in code and traces cannot be reconstructed reliably, and a recovered model cannot provide assurance equivalent to a validated forward model. This is especially consequential in regulated, safety-critical, or certification settings, where original requirements, rationale, and provenance form part of the assurance case. Recovered models can support comprehension and heuristic defect discovery in the middle ground

scoped by this paper, but they are not substitutes for upfront specifications and traceable assurance.

**Model Choice and Abstraction Level.** Different users need different models. A non-expert creator may need a domain model and a few critical invariants. A developer may need architectural and behavioral views. An AI repair agent may need machine-readable constraints and traceability to code. The next step is to compare model types such as class, component, sequence, state, data-flow, deployment, and constraint views for concrete decisions in the feedback loop. Some cases can reuse established UML or architectural notations. Others may need lighter, formalized, partial, or conversational forms of modeling that expose only the decisions at stake [15, 22].

**Verification and Repair.** Models can surface risks that are hard to see in raw code, including missing authorization paths, inconsistent schema use, architectural erosion, and model smells [25, 27, 37, 38]. This promise creates a responsibility to verify the models themselves. A wrong model can produce misplaced confidence. Verification should therefore check recovered models against code, tests, runtime traces, documentation, blueprints, and developer intent where available. Repair support should then translate model-level findings into concrete artifacts, including regression tests, static checks, repair prompts, pull request comments, or model transformations that guide later generation.

**Usability for Non-Experts.** The democratizing appeal of vibe coding should not be lost by reintroducing models in a form that only experts can use. Prior work shows that practitioners often use UML selectively and informally rather than as a complete formal language [10, 21, 33]. Model-mediated AI development should build on this reality [6]. Useful interfaces may combine lightweight diagrams, natural-language explanations linked to model elements, progressive disclosure of detail, and navigation from a model-level concern to the code or test that implements it. The research problem is not only notation design, but also when and how to present a model so it supports judgment without interrupting creative flow.

**Process Integration.** Models will matter only if they fit into the workflows where AI-generated software is produced. They should appear at moments when they help users make decisions, such as before deployment, after agentic code changes, during security review, or when a prototype gains real users. This requires tool support that connects modeling environments, IDEs, AI coding assistants, test systems, and repositories [35]. One practical direction is to treat model recovery as a CI or repository service that produces a compact review artifact whenever an agent changes code or whenever generated software crosses a deployment threshold. Recent advances in web-based modeling tool development [7, 17, 20] should facilitate the integration of model generation, rendering, and analysis into the AI-driven software engineering workflow.

**Evaluation.** This proposal needs empirical evaluation beyond plausible examples. Studies should ask whether recovered models help users identify missing authorization, invalid data ownership, architectural boundary violations, or unsafe deployment assumptions faster and more accurately than code review alone. They

should also test whether models improve comprehension of generated systems, support maintainability and evolution tasks, reduce unnecessary inspection effort, and increase justified confidence in AI-assisted changes. Evaluations should compare expert developers, junior developers, and non-expert creators, because the useful level of abstraction will likely differ across groups. Such studies would turn the proposal from a conceptual reframing into measurable evidence about when software models improve AI-assisted engineering.

## 5 Conclusion

Vibe coding and agentic software development illustrate both the promise and the risk of generative AI in software engineering. They make software creation more accessible, but they also allow prototypes to become consequential systems before their structure, assumptions, and risks are fully understood. We have argued that software models can help address this gap by taking on a renewed mediating role. In the agentic loop, recovered and refined models give AI systems symbolic feedback for checking, repair, and future constraints. In the human reflective loop, the same models give creators and developers inspectable representations of assumptions that would otherwise remain buried in prompts and generated code.

This is a bridge between software engineering practice and model-based reasoning. It does not require a full return to upfront modeling, nor does it treat models as static documentation. It treats them as working artifacts that connect intent, code, validation, and evolution. If vibe coding and agentic development are becoming a new form of prototyping, software models may be the artifacts that help turn generated prototypes into systems that can be understood, hardened, and maintained.

## Acknowledgments

This research was partially funded by the Austrian Science Fund (FWF) under grant 10.55776/PIN3275223 (<https://doi.org/10.55776/PIN3275223>) and by the Austrian Research Promotion Agency (FFG) through the Smart GLSP project (grant FO999925707).

## 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] 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
- [4] Nelly Bencomo, Jordi Cabot, Marsha Chechik, Betty H. C. Cheng, Benoit Combemale, Andrzej Wąsowski, and Steffen Zschaler. 2024. Abstraction Engineering. arXiv:2408.14074 [cs.SE] <https://arxiv.org/abs/2408.14074>
- [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] Dominik Bork, Stefan Klikovits, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2025. Inclusive Model-Driven Engineering for Accessible Software. In *28th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2025, Grand Rapids, MI, USA, October 5-10, 2025*. IEEE, 253–259. doi:10.1109/MODELS67397.2025.00030

- [7] Dominik Bork, Philip Langer, and Tobias Ortmayr. 2023. A Vision for Flexible GLSP-Based Web Modeling Tools. In *The Practice of Enterprise Modeling - 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28 - December 1, 2023, Proceedings (Lecture Notes in Business Information Processing, Vol. 497)*, João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper (Eds.). Springer, 109–124. doi:10.1007/978-3-031-48583-1\_7
- [8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice*. Morgan & Claypool Publishers.
- [9] 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)*.
- [10] Juan de Lara and Esther Guerra. 2025. Adaptive Modelling Languages: Abstract Syntax and Model Migration. *ACM Trans. Softw. Eng. Methodol.* 34, 3 (2025), 66:1–66:54. doi:10.1145/3702975
- [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 Esposito, 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] 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
- [17] Esther Guerra and Juan de Lara. 2018. On the Quest for Flexible Modelling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, Andrzej Wasowski, Richard F. Paige, and Øystein Haugen (Eds.). ACM, 23–33. doi:10.1145/3239372.3239376
- [18] 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
- [19] 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>
- [20] Andreas Hell, Martin Fleck, Philip Langer, and Dominik Bork. 2026. Realizing a Model Context Protocol (MCP) Server for the Graphical Language Server Platform (GLSP). In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS'2026)*. doi:10.1145/3822455.3830317
- [21] Robbert Jongeling and Federico Ciccozzi. 2024. Flexible Modelling: a Systematic Literature Review. *J. Object Technol.* 23, 3 (2024), 1–14. doi:10.5381/JOT.2024.23.3. A3
- [22] Robbert Jongeling, Federico Ciccozzi, Antonio Cicchetti, and Jan Carlson. 2022. From Informal Architecture Diagrams to Flexible Blended Models. In *Software Architecture - 16th European Conference, ECSA 2022, Prague, Czech Republic, September 19-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13444)*, Ilias Gerostathopoulos, Grace A. Lewis, Thais Vasconcelos Batista, and Tomás Bures (Eds.). Springer, 143–158. doi:10.1007/978-3-031-16697-6\_10
- [23] 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
- [24] 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).
- [25] Mohammed Mishbaudhin 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
- [26] 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.
- [27] Haris Mumtaz, Mohammad R. Alshayeb, Sajjad Mahmood, and Mahmood Ni-azi. 2019. A survey on UML model smells detection techniques for software refactoring. *J. Softw. Evol. Process.* 31, 3 (2019). doi:10.1002/SMR.2154
- [28] 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
- [29] 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.
- [30] 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
- [31] Anh Nguyen-Duc, Beatriz Cabrero Daniel, Adam Przybyłek, 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
- [32] Marten Oltröge, 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
- [33] Marian Petre. 2013. UML in Practice. In *35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 722–731. doi:10.1109/ICSE.2013.6606618
- [34] James Prather, Brent N. Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S. Randle, 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
- [35] Daniel Russo. 2024. Navigating the Complexity of Generative AI Adoption in Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 33, 5 (2024), 135:1–135:50. doi:10.1145/3652154
- [36] Phil Simon. 2022. *Low-Code/No-Code: Citizen Developers and the Surprising Future of Business Applications*. Racket Publishing.
- [37] Muhamed Smajevic and Dominik Bork. 2021. From Conceptual Models to Knowledge Graphs: A Generic Model Transformation Platform. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*. IEEE, 610–614. doi:10.1109/MODELS-C53483.2021.00093
- [38] Muhamed Smajevic, Simon Hacks, and Dominik Bork. 2021. Using Knowledge Graphs to Detect Enterprise Architecture Smells. In *The Practice of Enterprise Modeling - 14th IFIP WG 8.1 Working Conference, PoEM 2021, Riga, Latvia, November 24-26, 2021, Proceedings (Lecture Notes in Business Information Processing, Vol. 432)*, Estefania Serral, Janis Stirna, Jolita Ralyté, and Janis Grabis (Eds.). Springer, 48–63. doi:10.1007/978-3-030-91279-6\_4
- [39] Paul Swail. 2022. When Prototype Becomes Production. <https://serverlessfirst.com/emails/when-prototype-becomes-production>. Accessed: 2025-09-29.
- [40] Eric Von Hippel. 2006. *Democratizing innovation*. the MIT Press.
- [41] 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.
- [42] Yun Wang and Yan Lu. 2025. Interaction, Process, Infrastructure: A Unified Architecture for Human-Agent Collaboration. *arXiv preprint arXiv:2506.11718* (2025).
- [43] Matt Welsh. 2023. The End of Programming. *Commun. ACM* 66, 1 (2023), 34–35. doi:10.1145/3570220
- [44] Shai Yallin. 2021. Permanent Prototypes. <https://www.shaiyallin.com/post/permanent-prototypes>. Accessed: 2025-09-29.