Re-Imagining Software Visualization for Spec-Driven Development

Giovanni Rosa
Universidad Rey Juan Carlos
Madrid, Spain
giovanni.rosa@urjc.es

Mario Linares-Vásquez
Universidad de los Andes
Bogotá, Colombia
m.linaresv@uniandes.edu.co

Gregorio Robles
Universidad Rey Juan Carlos
Madrid, Spain
grex@gsyc.urjc.es

***Abstract*—Spec-Driven Development (SDD) is reshaping AI-assisted software engineering by elevating natural-language specifications to first-class artifacts that developers author, review, and evolve as deliberately as code. Yet specifications are still handled as flat, unstructured text, even as they grow into large, densely interconnected repositories of requirements, design decisions, and constraints. This is precisely the comprehension problem that motivated software visualization for source code two decades ago. In this paper, we argue that visualization metaphors should not stop at the code boundary and propose a framework to visualize SDD projects. We organize the framework into five levels of techniques, progressing from understanding the structure and quality of individual specifications, to their internal relationships, their traceability to generated code, their coevolution over time, and finally integrated workspaces that support complex tasks. We also outline the open research challenges each level raises, sketching a research agenda for the community.**

***Index Terms*—Spec-Driven Development, SDD, Software Visualization, Software Comprehension, Software Evolution**

I. Introduction

Spec-Driven Development (SDD) is an emerging paradigm in the era of AI-agent programming [1]. Its core idea is to replace ad-hoc model prompting with a more formal way of expressing natural-language instructions, using a set of specification files [2] that describe the different parts of a system. The SDD workflow is supported by several coding tools, such as *Claude Code*[1], *Kiro*,[2] and *GitHub Spec Kit*.[3]

As software development shifts toward spec-driven, AI-assisted workflows, the specification itself is becoming a first-class artifact that developers read, edit, and reason about as often as the code it generates—yet it is still treated as plain, unstructured text. This is the same blind spot that motivated code visualization two decades ago. Just as a source tree of thousands of files quickly overwhelms a developer's ability to hold its structure in mind, a growing specification saturates comprehension long before it saturates the page: it has its own hierarchy of requirements, its own cross-references and dependencies between components, and its own constraints that ripple across the system. In fact, a specification has its own "files" (requirements, user stories, interface contracts), its own "folders" (modules, features, subsystems), and its own dependency graph (which requirements constrain which others, which generated code satisfies which spec, and where a change in the spec ripples into the implementation). The techniques developed for code were built precisely to make large, interlinked, hard-to-navigate structures legible at a glance, and there is no principled reason they should stop at the code boundary. Extending visualization from code to specifications is therefore not a stylistic choice, but a natural continuation of the argument that motivated visualizing code in the first place: once an artifact grows large and densely connected enough, visual metaphors become not a convenience but a necessity for comprehension.

In this vision paper, we propose a framework envisioning how software visualization adapts and evolves for SDD projects, covering three main goals: (i) to describe the different SDD artifacts and its relations through different visualization metaphors; (ii) to provide traceability between specification artifacts and code; and (iii) to support design, implementation, maintenance and evolution activities (*e.g.,* when a bug appears, tracing it back to the specification that produced it).

II. Background and Related Work

Spec-Driven Development (SDD) is an emerging approach to AI-assisted software engineering in which a structured specification—rather than naive model prompting, *i.e.,* "conversational development"—guides the development process. The concept, rooted in Agile methodology [3], aims to make requirements and decisions persistent and reviewable, mainly to (i) provide a formal definition of requirements and constraints, (ii) improve reproducibility, and (iii) enable collaboration among different contributors or AI agents.

One of the most popular SDD frameworks nowadays is *GitHub Spec Kit*, which provides the most formal implementation of a structured SDD workflow (as of today): its core is the `constitution.md` file, which encodes the project's core principles and practices, followed by a sequence of specification, plan, task, and implementation artifacts. The main goal of these artifacts is to separate the *what* and *why* of a change from the *how*. A typical workflow is an iterative sequence of phases that moves from higher-level requirements to a set of actionable tasks, namely: specifying requirements, planning, defining tasks, generating the implementation, and, optionally, testing and verifying.

[1]https://code.claude.com/docs
[2]https://kiro.dev/
[3]https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/

There are other alternative tools: First, *Claude Code*, an agentic coding environment that relies on context files (*i.e.,* `CLAUDE.md`) that can implement SDD. Second, *OpenSpec*[4], which offers a change-oriented workflow, instead.

Despite their differences, these tools (*i.e., Spec Kit*, *Claude Code*, *OpenSpec*) share the principle that explicit and reviewable intent, defined through specification artifacts, formalizes and persists the interaction between developers and AI agents.

Regarding previous efforts of the research community, interest in SDD and agentic coding is growing rapidly. For instance, recent work frames autonomous coding agents as emerging "teammates" that are reshaping software engineering practice [4], and characterizes SDD as a shift from writing code to writing contracts in the age of AI coding assistants [5]. Moreover, empirical studies have begun to examine how developers author and use agentic coding manifests and repository-level context files in practice [6], [7], observing that such files are still largely hand-written and of uneven quality. Other studies report early experiences applying spec-driven workflows to domain-driven development [2] and extending them to multi-agent settings [8]. **Despite this momentum, to the best of our knowledge no prior work investigates how to *visualize* SDD artifacts and projects.**

III. A Framework for SDD Visualization

In Spec-Driven Development (SDD), textual specifications serve as the primary artifacts and central knowledge repository throughout the software development lifecycle. Rather than merely outlining functional requirements, these specifications comprehensively document architectural decisions, design principles, workflows, and implementation guidelines across various project files. Ultimately, they provide the core framework that governs and coordinates all software development activities for both human developers and AI assistants.

While most software visualization techniques and metaphors have traditionally focused on source code artifacts [9]–[13], relatively little attention has been paid to visualization of specification-centric artifacts. Furthermore, establishing and maintaining traceability links between specifications and code remains a longstanding challenge in software engineering [14]–[16]. As specifications become the primary artifact in SDD, software visualization research must evolve to support this paradigm shift. We envision a new framework of visualization techniques/metaphors that enable software engineers to understand the structure, semantics, and evolution of specifications, their relationships with generated code and other software artifacts, and the coevolution of these artifacts throughout the software development lifecycle. Such visualizations should also support specialized software engineering activities within SDD frameworks, including specification authoring, quality assessment, traceability analysis, maintenance, and evolution.

Motivated by this vision, we propose a framework for SDD software visualizations that outlines a research agenda for both researchers and practitioners. The framework is organized into five levels of visualization techniques, progressing from foundational capabilities to increasingly sophisticated forms of analysis and interaction (Figure 1). We adopt a level-based perspective because we view SDD as an emerging software engineering paradigm whose practices, tools, and development frameworks are still evolving. Consequently, several foundational research challenges remain open, including the definition of specification quality metrics, specification smells and anti-patterns, human factors in SDD, and techniques for analyzing the coevolution of specifications and generated artifacts. As SDD frameworks continue to mature and gain adoption, we expect these levels of visualization techniques to evolve accordingly, providing increasingly powerful support for AI-assisted software engineering.

A. First Level

Specifications are composed of multiple entities and heterogeneous information. Consequently, a single specification file may contain sections describing different organizational, design, and implementation (ODI) aspects of a software system. Furthermore, an SDD project typically comprises multiple types of specification files, each serving a distinct purpose in supporting software development. Although these files are expected to have well-defined boundaries, ODI aspects may still be scattered across multiple files and sections, similarly to the way concerns are often distributed across source code and traditional software documentation.

Early software visualization techniques employed geometric metaphors (*e.g.,* Code City-style [17], [18] visualizations) and text-based representations (*e.g.,* word clouds and concept maps) to help software engineers understand software artifacts. Inspired by these approaches, the first level of SDD visualizations should focus on helping engineers explore and assess specification artifacts before code generation. Specifically, these visualizations should support: (i) understanding the structure, organization, and intent of specification files; (ii) assessing the quality of specifications through dedicated quality indicators and metrics; and (iii) identifying specification-level issues before AI agents generate code for feature implementation or maintenance tasks.

Achieving this vision requires advances beyond visualization techniques alone. In particular, the research community must develop specification-oriented quality metrics, as well as taxonomies of specification bugs, spec smells, anti-patterns, and other specification quality issues. These foundational artifacts would not only enrich future visualizations but also provide actionable insights that improve specification quality and ultimately enhance the reliability of AI-assisted software engineering.

B. Second Level

In SDD, specification files serve multiple purposes, including defining domain models, implementation guidelines, organizational principles, quality attributes, and functional and non-functional requirements. Consequently, the second level

[4]https://github.com/Fission-AI/OpenSpec

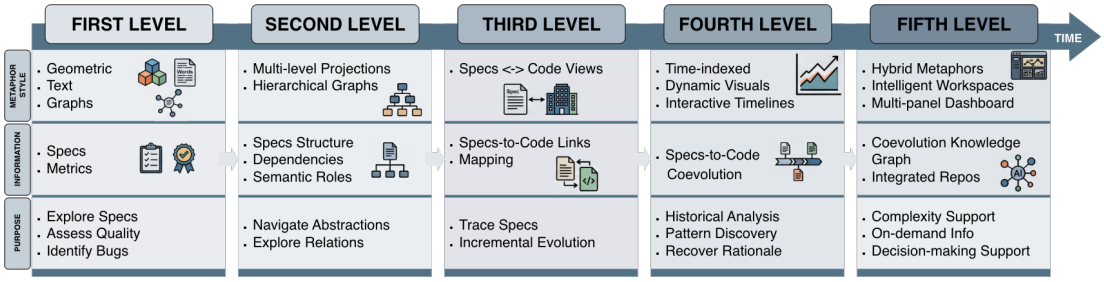


Fig. 1. A summary of the proposed framework for Spec-Driven Development visualization. Image created by the authors with the assistance of AI tools.

of SDD visualizations should move beyond simple document representations toward richer visualizations that enable software engineers to explore and understand the complex relationships among specification entities. These relationships include dependencies, associations, similarities, composition and inclusion hierarchies, inheritance, semantic roles, and other structural or conceptual connections.

Although specification files may contain explicit references to one another, such links are often incomplete or entirely absent. Therefore, this level of visualizations should leverage Natural Language Processing (NLP), Information Retrieval (IR), and knowledge extraction techniques to automatically identify entities, topics, relationships, concerns, and semantic structures embedded within specification documents.

Furthermore, software specifications naturally organize information across multiple abstraction dimensions, such as viewpoints, stakeholder concerns, architectural layers, roles, and quality attributes. This characteristic creates an opportunity to design multi-level and multi-perspective visualizations that allow software engineers to navigate specification artifacts at different levels of abstraction. Such visualizations should support interactive, on-demand exploration, enabling users to dynamically focus on the information most relevant to their current development, maintenance, or analysis task.

C. Third Level

Specifications are ultimately materialized into code-level artifacts by both AI agents and human developers, bringing renewed attention to the longstanding challenge of establishing traceability links between high-level artifacts (*e.g.,* specifications and documentation) and low-level artifacts (*e.g.,* source code). In this context, visualizations that enable software engineers to understand how prompts and specifications are translated into generated code across multiple levels of abstraction could significantly improve software comprehension. Such visualizations would also facilitate the analysis of the coevolution of specifications and code, as well as the assessment of different AI-assisted code generation strategies. This understanding is essential for supporting incremental software evolution and maintenance, as advocated by the software evolution community [19], while addressing the unique characteristics of AI-assisted development environments.

However, visualizing traceability relationships remains a challenging research problem [20]. Consequently, future visualization techniques should provide specialized support for recovering, discovering, and exploring specification-to-code traceability links at multiple levels of granularity. These visualizations should also adapt to developers' on-demand information needs [16], allowing engineers to seamlessly navigate between strategic, architectural, and implementation perspectives. Furthermore, because source code becomes a central artifact in this level of visualizations, scalability, responsiveness, and interactive performance emerge as critical quality attributes that directly influence the usability and effectiveness of visualization tools.

D. Fourth Level

While previous levels of software visualizations primarily focused on static representations, the coevolution of specifications and source code requires visualizations that support historical analysis and traceability recovery. Consequently, fourth-level software visualizations should emphasize time-aware representations and pattern discovery to help engineers understand how requirements, specifications, and code evolve together over time.

Achieving this vision, however, requires more explicit evidence of traceability relationships. In traditional software development, developers often leave explicit traceability clues through code comments, commit messages, issue trackers, and pull requests that connect implementation artifacts with higher-level requirements and design specifications. This assumption does not necessarily hold in AI-assisted software development. Autonomous coding agents generate code primarily from prompts and specifications, while exercising a certain degree of autonomy during code generation. As a result, the rationale linking generated code to its originating prompts, specifications, and intermediate reasoning is often lost.

To improve transparency, AI-assisted development environments should explicitly expose the relationships between generated code and the prompts/specifications that guided its creation. Since current coding agents do not provide this capability by default, these traceability links could be captured through explicit instructions embedded in prompts/specifications (*e.g.,* by using CoT prompting), or generated automatically by the agents themselves. The availability of such AI-generated traceability information, combined with existing traceability recovery techniques, would enable researchers to design novel metaphors, interactive visualizations, and analysis tools that support the exploration and understanding of the coevolution of specifications, prompts, and software artifacts throughout the software development lifecycle.

E. Fifth level

Once the previous levels have matured, software visualization research should advance toward fifth-level visualizations that support complex and knowledge-intensive software engineering activities. Rather than focusing solely on artifact exploration, these visualizations should assist engineers in understanding, reasoning about, and evolving large-scale software systems. For example, evolving the architecture of a complex system requires a comprehensive understanding of user requirements, stakeholder concerns, architectural constraints, and technology choices, while simultaneously analyzing how source code embodies specifications, design decisions, and implementation rationale. Supporting these activities demands integrated, continuously updated, and easily navigable representations of software artifacts.

Fifth-level visualizations should provide unified and interactive views that enable software engineers to understand, navigate, browse, and, when appropriate, directly edit: (i) specifications and source code, including their entities, structures, dependencies, and quality indicators; (ii) traceability relationships among artifacts spanning multiple levels of abstraction and implementation; and (iii) the coevolution of specifications, prompts, code, tests, documentation, and other development artifacts throughout the software lifecycle.

Beyond artifact exploration, these visualizations should actively support software engineering tasks by providing contextual, on-demand information tailored to developers' immediate needs, without requiring complex configuration or extensive setup. They should seamlessly integrate into AI-assisted development environments while maintaining high levels of scalability, responsiveness, usability, user experience, and interactive performance, even for large-scale projects developed by globally distributed teams. Ultimately, fifth-level visualizations should function not merely as visualization tools but as intelligent workspaces that facilitate comprehension, decision-making, collaboration, and continuous software evolution in SDD environments.

IV. Conclusion

Spec-Driven Development turns natural-language specifications into the primary artifact of AI-assisted software engineering. In this paper, we argued that the software visualization techniques developed for source code should be extended to specifications, and we proposed a roadmap organized into five levels of visualization techniques spanning from the aspects related to the individual specifications, to the coevolution and integration in complex engineering tasks.

Realizing this vision requires progress on foundational problems that go beyond visualization itself, including specification quality metrics, catalogs of spec smells and anti-patterns, human factors in SDD, and mechanisms that make spec-to-code traceability explicit in agentic development environments. We hope this roadmap encourages both researchers and practitioners to treat specifications as a first-class subject of software visualization, and to build the metaphors and tools that AI-assisted software engineering will increasingly demand.

References

[1] B. Böckeler. (2025) Understanding spec-driven-development: Kiro, spec-kit, and tessl. [Online]. Available: https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html

[2] J. de Barros Santos, "A spec-driven workflow for ai-assisted domain-driven development: Insights from practice," in *Proceedings of the 3rd International Workshop on Large Language Models For Code*, 2026, pp. 131–134.

[3] J. S. Ostroff, D. Makalsky, and R. F. Paige, "Agile specification-driven development," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 104–112.

[4] H. Li, H. Zhang, and A. E. Hassan, "The rise of ai teammates in software engineering (se) 3.0: How autonomous coding agents are reshaping software engineering," *arXiv preprint arXiv:2507.15003*, 2025.

[5] D. B. Piskala, "Spec-driven development: From code to contract in the age of ai coding assistants," *arXiv preprint arXiv:2602.00180*, 2026.

[6] W. Chatlatanagulchai, K. Thonglek, B. Reid, Y. Kashiwa, P. Leelaprute, A. Rungsawang, B. Manaskasemsak, and H. Iida, "On the use of agentic coding manifests: An empirical study of claude code," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2025, pp. 543–551.

[7] T. Gloaguen, N. Mündler, M. Müller, V. Raychev, and M. Vechev, "Evaluating agents. md: Are repository-level context files helpful for coding agents?" *arXiv preprint arXiv:2602.11988*, 2026.

[8] P. Taghavi and S. Bhavani, "Spec kit agents: Context-grounded agentic workflows," *arXiv preprint arXiv:2604.05278*, 2026.

[9] L. Bedu, O. Tinh, and F. Petrillo, "A tertiary systematic literature review on software visualization," in *2019 Working Conference on Software Visualization (VISSOFT)*, 2019, pp. 33–44.

[10] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 7, pp. 913–933, 2011.

[11] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, 2003.

[12] A. R. Teyseyre and M. R. Campo, "An overview of 3d software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 87–105, 2009.

[13] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, and W. Chen, "A systematic literature review of modern software visualization," *Journal of Visualization*, vol. 23, no. 4, pp. 539–558, Aug. 2020. [Online]. Available: https://doi.org/10.1007/s12650-020-00647-w

[14] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1199–1210.

[15] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota, M. Lanza, and D. C. Shepherd, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 590–601. [Online]. Available: https://doi.org/10.1145/3377811.3380405

[16] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand developer documentation," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 479–483.

[17] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *2007 4th IEEE International workshop on visualizing software for understanding and analysis*. IEEE, 2007, pp. 92–99.

[18] D. Moreno-Lumbreras, J. M. Gonzalez-Barahona, G. Robles, and V. Cosentino, "The influence of the city metaphor and its derivates in software visualization," *Journal of Systems and Software*, vol. 210, p. 111985, 2024.

[19] V. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Softw.*, vol. 21, no. 4, p. 62–69, Jul. 2004. [Online]. Available: https://doi.org/10.1109/MS.2004.17

[20] X. Chen, J. Hosking, and J. Grundy, "Visualizing traceability links between source code and documentation," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2012, pp. 119–126.