Aligning Model-Driven Engineering and Spec-Driven Development for Agentic AI Software Engineering

Jordi Cabot[*], Benoit Combemale[**], Jean-Michel Bruel[†], Robert Clarisó[‡], Juan de Lara[§], Davide Di Ruscio[⋆], and Gunter Mussbacher[⊕]

[*]LIST & University of Luxembourg
[**]Inria & University of Rennes, France
[†]University of Toulouse, France
[‡]Universitat Oberta de Catalunya, Spain
[§]Autonomous University of Madrid, Spain
[⋆]Università degli Studi dell'Aquila, Italy
[⊕]McGill University, Canada

ABSTRACT Software development is becoming increasingly AI-driven, raising questions about the role of traditional software engineering in this new paradigm. Exploring how advanced AI tools can enhance professional development practices beyond casual use, such as *vibe coding*, is our responsibility as software engineers. In this paper, we synthesize existing concepts in *spec-driven development* (SDD) and *model-driven engineering* (MDE) to propose a framework that treats specifications as abstract models, even if expressed in free-form natural language. We argue that integrating free-form and structured natural language specifications with model-driven engineering can improve the impact of AI agents in software development, making them more accessible and effective alongside human developers. To enable effective interplay between humans and agents, we foresee a three-step integration, starting from a simple interoperability bridge to embedding each other's best practices to finally a unified single development platform for everybody.

KEYWORDS Agentic Software Development, Spec-Driven Development, Model-Driven Engineering

1. Introduction

Advances in generative artificial intelligence – especially large language models (LLMs) – are reshaping many disciplines, including software engineering. Both general-purpose LLMs (like those of the GPT family, Gemini, Llama, or Mistral) and code-oriented LLMs (like Claude, Codex, Code Llama, or StarCoder) have shown remarkable usefulness for programming tasks like code generation, code understanding, or test case synthesis (M. Chen et al. 2021; Jiang et al. 2026; Schäfer et al. 2024).

JOT reference format:
Jordi Cabot, Benoit Combemale, Jean-Michel Bruel, Robert Clarisó, Juan de Lara, Davide Di Ruscio, and Gunter Mussbacher. *Aligning Model-Driven Engineering and Spec-Driven Development for Agentic AI Software Engineering.* Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.2026.25.2.c1

In recent years, the use of LLMs for software development has evolved beyond prompting general-purpose conversational systems such as ChatGPT or Gemini, which programmers initially employed for coding assistance. These models have since been incorporated into integrated development environments (IDEs) as embedded programming assistants. Representative systems include GitHub Copilot (GitHub 2021), Tabnine (Tabnine 2026), Amazon Q Developer (Amazon Web Services 2024), and Caret (Contreras et al. 2026), among others. Within these environments, LLM-based assistants support developers by automating tasks such as code completion from natural language prompts, documentation generation, and code refactoring. These assistants accelerate the development process, while being aware of the development context (Ziegler et al. 2024).

Still, more recently, agentic AI emerged as an enabler of an even higher level of automation. LLM-based agents (*coding agents*, e.g., Claude Code (Anthropic 2024)) use a ReACT

prompt-style loop (Yao et al. 2023) to be able to use external tools to interact with the environment. When applied to software development, agents are provided with an extensible set of tools, e.g., file system tools (to read or create files), terminal/shell execution tools, compilers, and static analysis tools, among many others (Wang et al. 2025). These agents, often available via web platforms – like Lovable (Lovable 2026), Base44 (Base44 2026), or Bolt.new (StackBlitz 2026) – allow users with no technical profile to create software just by using natural language. This way, the term *vibe coding* emerged to describe code generated from improvised prompts and chat-based interactions with software agents (Karpathy 2025; Meske et al. 2025), e.g., via Cursor (Anysphere Inc. 2022).

Today, the software industry is increasingly adopting AI-based development (Apostolou et al. 2026; Jellyfish 2025).[1] Processes and practices to make AI-based development fit for professional development – beyond vibe coding – are starting to emerge. This trend is known as *spec-driven development* (SDD) (Delimarsky 2025), and its idea is to base development on specifications: natural language descriptions of functionality, non-functional requirements, and architectural constraints, often encoded in Markdown files, that guide agents in their software construction tasks. Contemporary tools such as OpenAI's Codex (OpenAI 2025), GitHub Spec Kit (GitHub 2025), OpenSpec (Fission AI 2025), or Kiro (Amazon Web Services 2025) can now autonomously generate, modify, and validate software artifacts from high-level instructions.

All these technologies mentioned above exemplify an increasingly rich ecosystem of AI-supported development environments, ranging from conversational assistants that react to developer prompts to more autonomous agentic systems capable of planning and executing development tasks.

On reflection, model-driven engineering (MDE) (Brambilla et al. 2017) shares many principles with SDD, in particular the goal of automation, and the use of interrelated artifacts – with a high-level of abstraction – to build the final system, often by code generation. As key industry players are investing heavily in AI-based automation with agents, one may wonder in this current context: *"What will be the role and relevance of model-driven engineering for software development?"*

While LLMs have been applied to MDE tasks, e.g., the generation of domain models (K. Chen et al. 2023; Cámara et al. 2023; Mercado et al. 2025), goal models (B. Chen et al. 2023), and sequence diagrams (Jahan et al. 2024), but also for model completion (Chaaben et al. 2023; Almonte et al. 2026), the integration of MDE and SDD has not yet been explored in greater detail (Feng et al. 2026). The goal of this paper is to align SDD and MDE by reflecting on the fact that SDD specifications are actually models, but expressed in free-form and structured natural language. Hence, building upon this idea, we propose a structured vision for integrating MDE and SDD. In particular, we claim that while SDD artifacts target agents, these – often long and repetitive – textual artifacts may be hard to digest by human developers. Moreover, SDD specifications can cover aspects that might be contributed by different stakeholders with different expertise. This context requires proper separation of concerns, abstraction engineering, and support for analysis and design space exploration, which MDE can provide. Hence, abstracting SDD specifications as models – and providing a bidirectional bridge between them – would make SDD a more human-centered paradigm, enabling a more natural collaboration between teams of humans and agents. To this end, we propose a conceptual model that enables the integration of MDE and SDD to further support such collaboration.

The rest of this paper is organised as follows. Section 2 exposes the problem statement. Section 3 describes the envisioned interplay between MDE and SDD, and Section 4 ends discussing future plans.

2. Problem Statement

The capabilities of a coding agent are achieved through the combination of two elements: an LLM and a *harness*. The LLM serves as the underlying engine, executing tasks based on the instructions and context. Meanwhile, the harness is a software framework that envelops the LLM to steer LLM output into the right direction. It provides features such as an overall strategy, problem decomposition into subgoals and tasks, subtask orchestration, knowledge-base management (short and long-term memory), context management (retrieval-augmented generation, context compression), and access to external APIs (model-context protocol (MCP) (Model Context Protocol Contributors 2025)), tools, and resources (file system access, web browsing, code execution) regulated by user-defined privileges. Moreover, agent capabilities and behavior can be customized and extended by the user in several ways, either by providing system-level prompts (often described using terms such as "soul", "constitution", or "steering"), defining programmed actions triggered by development events and milestones (hooks), or creating or importing reusable workflows (skills) that can include code execution or the usage of external tools. Finally, the harness may also provide *observability* features to let the user understand, monitor, and control the coding agent's activities.

While vibe coding (see Figure 1) is highly effective for exploratory prototyping, such approaches often suffer from limited traceability, architectural drift, and weak support for large-scale projects. In response, the SDD harness introduces explicit intermediate artifacts (typically structured Markdown specifications such as requirements, design descriptions, task decompositions, or steering files) that become the primary medium through which human intent is captured and subsequently interpreted by agents. In systems such as Kiro, a feature request may be transformed into a sequence of specification artifacts (e.g., requirements.md, design.md, and tasks.md). Each one represents a complementary view of the software development process, from analysis to design to implementation, co-existing as explicit models that are created, maintained, and kept consistent by specialized agents throughout the engineering lifecycle. In this sense, SDD shifts AI-assisted software development from spontaneous prompt-centric interactions to a more explicit, model-centric process. In this process, specifications serve as persistent, machine-manipulable artifacts that drive various agents.

[1] https://survey.stackoverflow.co/2025/ai

For decades, MDE has advocated a development lifecycle centered around models as first-class engineering artifacts. Through abstraction engineering (Bencomo et al. 2024) and separation of concerns, MDE promotes the construction of models that make systems understandable and analyzable, and that can be transformed by explicitly representing different viewpoints of a system (Combemale et al. 2016). Models in MDE primarily serve human engineering activities: they support reasoning, analysis, communication, design exploration, verification/validation, traceability, and collaboration among stakeholders, while automated transformations leverage these models to derive downstream artifacts. The value of models in MDE, therefore, lies not only in automation but also in their role as shared cognitive and collaborative abstractions that bridge multiple engineering concerns.

Interestingly, SDD exhibits several structural similarities with this model-centric vision as illustrated by Figure 1. Like MDE, SDD organizes software development around explicit intermediate artifacts that progressively refine intent into implementation. Requirements specifications, design documents, task graphs, rule files, and contextual steering artifacts all function as abstractions that mediate between high-level intent and executable software. However, unlike MDE, SDD emphasizes these specifications as machine-grounding artifacts for agentic execution (e.g., harness engineering), optimized primarily for machine consumption rather than human cognition and separation of concerns. In practice, these artifacts are often maintained as lightweight Markdown files, which are particularly well suited to agentic workflows because they can be directly consumed, generated, and updated by AI agents. Yet, they also tend to be verbose, to mix multiple concerns within the same artifact, and to provide only limited support for explicit abstraction boundaries and separation of concerns. By contrast, models in the MDE sense are specifically designed to provide abstractions, viewpoints, and separation of concerns, thereby supporting structured reasoning over complex systems. Their drawback is that progressing from these abstractions toward executable artifacts often requires explicit and potentially sophisticated model transformations, which may introduce substantial engineering complexity. In other words, while MDE models are typically engineered to improve human understanding, modularity, and coordination, SDD artifacts are increasingly engineered as operational contexts for autonomous agents. This suggests a potentially complementary relationship between the two: SDD provides lightweight, agent-friendly representations for harness engineering, whereas MDE provides principled mechanisms for abstraction and separation of concerns, at the cost of a traditionally heavier transformation infrastructure.

This reveals a paradox. MDE and SDD both rely on model-centric development lifecycles, yet they embody complementary and largely disconnected assumptions. As a consequence, current AI-assisted software development often underutilizes decades of advances in MDE in abstraction engineering, traceability, viewpoint management, interoperability support, consistency preservation, system analysis, and design space exploration (see Figure 1). Furthermore, MDE employs deterministic transformations (M2M in Figure 1), while LLM-based generation is inherently non-deterministic (🤖 in Figure 1). Conversely, traditional MDE workflows do not yet fully exploit autonomous agents as active collaborators capable of reasoning over models and participating in engineering activities.

This gap motivates the need for a deeper integration between MDE and SDD. Rather than treating humans as prompt authors and agents as code generators, or treating models solely as documentation or transformation inputs, we envision a model-centric software engineering process in which humans and agents collaboratively co-engineer software through shared or aligned abstractions. Such an integration could elevate human-agent interaction to genuine human-agent collaboration, in which MDE abstractions provide rich, explicit, and structured representations for agents, and agents, in turn, augment human reasoning, exploration, and automation across the software development lifecycle. In this perspective, models become not merely artifacts manipulated by either humans or agents, but shared collaboration media that amplify the capabilities of both.

3. Envisioned Interplay of MDE and SDD

As stated above, MDE and SDD have strong parallelisms but target different users: human vs agents. Neither of the two has been designed to target mixed teams where humans and agents need to interact and collaborate, which is a real need once we move from vibe-coded apps to complex system development. To enable effective interplay between humans and agents (and between MDE and SDD), we propose a three-step integration, from a simple bridge to a single unified development platform that can be used by both.

3.1. Interoperability Bridge

The first step is to facilitate communication and interoperability among all team members, regardless of their role. Each team member works with the "concrete notation" they prefer, be it a Markdown file or any type of modeling representation. While it is true that humans can read Markdown files, this format is not ideal for human comprehension: the idiom "a picture is worth a thousand words" is particularly fitting here (in this case, "picture" refers to "a more abstract representation", i.e., broader than images). Similarly, while agents can understand images, this is not an optimal format for them and requires multimodal capabilities to process image content. Furthermore, some team members may require distinct abstractions, which do not need to be represented for other team members. This requires a bidirectional transformation between the two, which could be decomposed into the following components, and needs to consider that different but aligned abstractions may be required for SDD and MDE activities:

– A new *SDD metamodel* representing the common abstractions used in SDD files. This metamodel would enable the representation of different types of SDD files as instances of a generic metamodel, facilitating the implementation of MDE approaches across different agent families and simplifying the migration of a project to a different coding agent without requiring a "cold restart".

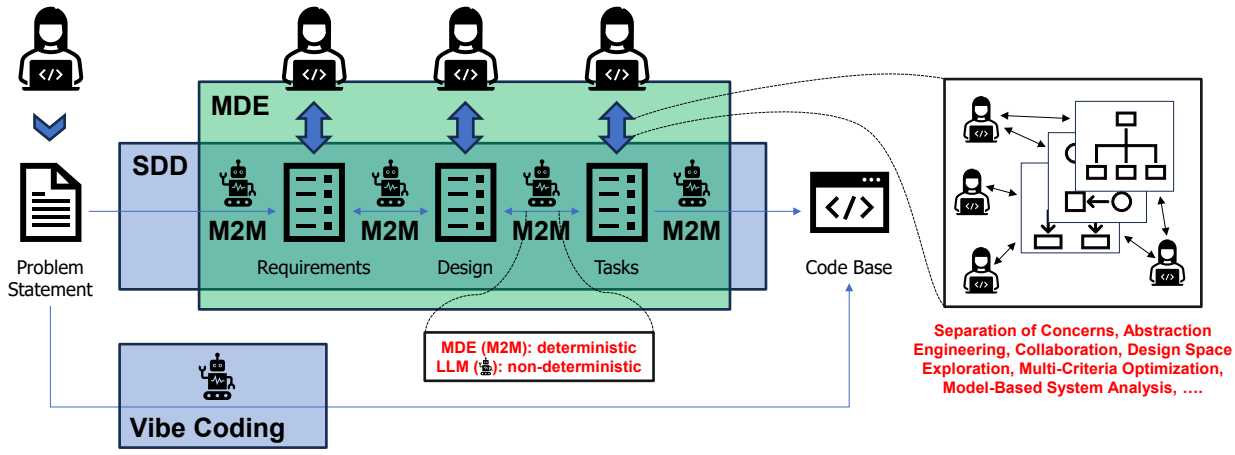

Figure 1 Commonalities and variabilities of vibe coding, SDD, and MDE

– A *text-to-model* (and the corresponding *model-to-text*) *transformation* from the Markdown file to the SDD metamodel (and vice-versa). As part of this transformation, we would need to parse natural language text and identify abstractions in the metamodel from free-form and structured textual descriptions. The accompanying text also needs to be stored as additional information to regenerate the original file at all times.
– A *model-to-model bidirectional transformation* from the SDD metamodel to the target modeling language (e.g., UML). This transformation may be only partial if some abstractions do not need to be shared between the two, as they are considered relevant only to one type of participant.

We analysed the specification mechanisms of three tool-supported SDD approaches: GitHub Spec Kit (GitHub 2025), Kiro (Amazon Web Services 2025), and OpenSpec (Fission AI 2025), as summarised in Table 1. As illustrated by Figure 1, all three approaches follow a similar, high level, four-stage process: requirements elicitation, system design, implementation planning, and subsequent code generation.

A key characteristic of these SDD approaches is that specification artifacts (e.g., Markdown documents) are not typically authored manually from scratch by developers. Instead, they are generated by the tool or agent based on high-level user inputs, often iteratively refined through interactive developer–agent exchanges. In this interaction loop, the developer provides initial goals, constraints, or partial descriptions, while the agent incrementally produces structured specification artifacts. The developer may then review, correct, or extend these outputs, leading to an iterative co-construction process rather than a one-shot specification phase.

In the requirements phase, all approaches are based on requirements lists and scenarios. Kiro and GitHub Spec Kit also use user stories, and GitHub Spec Kit includes lightweight domain models already in this phase. Meanwhile, in the design both Kiro and GitHub Spec Kit make heavy use of models (encoded as text in the Markdown), including domain models, data models, architecture diagrams in Mermaid format (Mermaid 2014), state machines, and contracts among others. Finally, before generating code, all approaches produce an implementation plan, made of a sequence of tasks with detailed information for each one.

Table 1 Specification mechanisms of SDD approaches

SDD	Requirements	Design	Impl. Plan
GitHub Spec Kit	Requirements List User stories Scenarios Domain model	Data model State machines Contracts	Task model
Kiro	Requirements list User stories Scenarios Glossary	Domain model Data model Architecture diagram Components description Test strategy	Task model
OpenSpec	Requirements list Scenarios	Goals, non-goals Risks	Task model

Figure 2 shows a consolidated metamodel that includes the elements found in each approach, divided into four packages: *SDD-core*, *specs*, *design*, and *implementation*. The figure uses both color codes and annotations to convey which approach uses each element. That is, the metamodel shows the comparative union of elements found in each approach. A pivot, unified metamodel is the subject of future research.

The *SDD-core* package shows that all approaches use requirements (`Spec`), design, and implementation plans. There are small differences, though. OpenSpec breaks upfront the requirements into several `Specs`, providing separate specifications for each. In addition, GitHub Spec Kit creates design and implementation plans specifically for each `Spec`, while for OpenSpec and Kiro these are global documents.

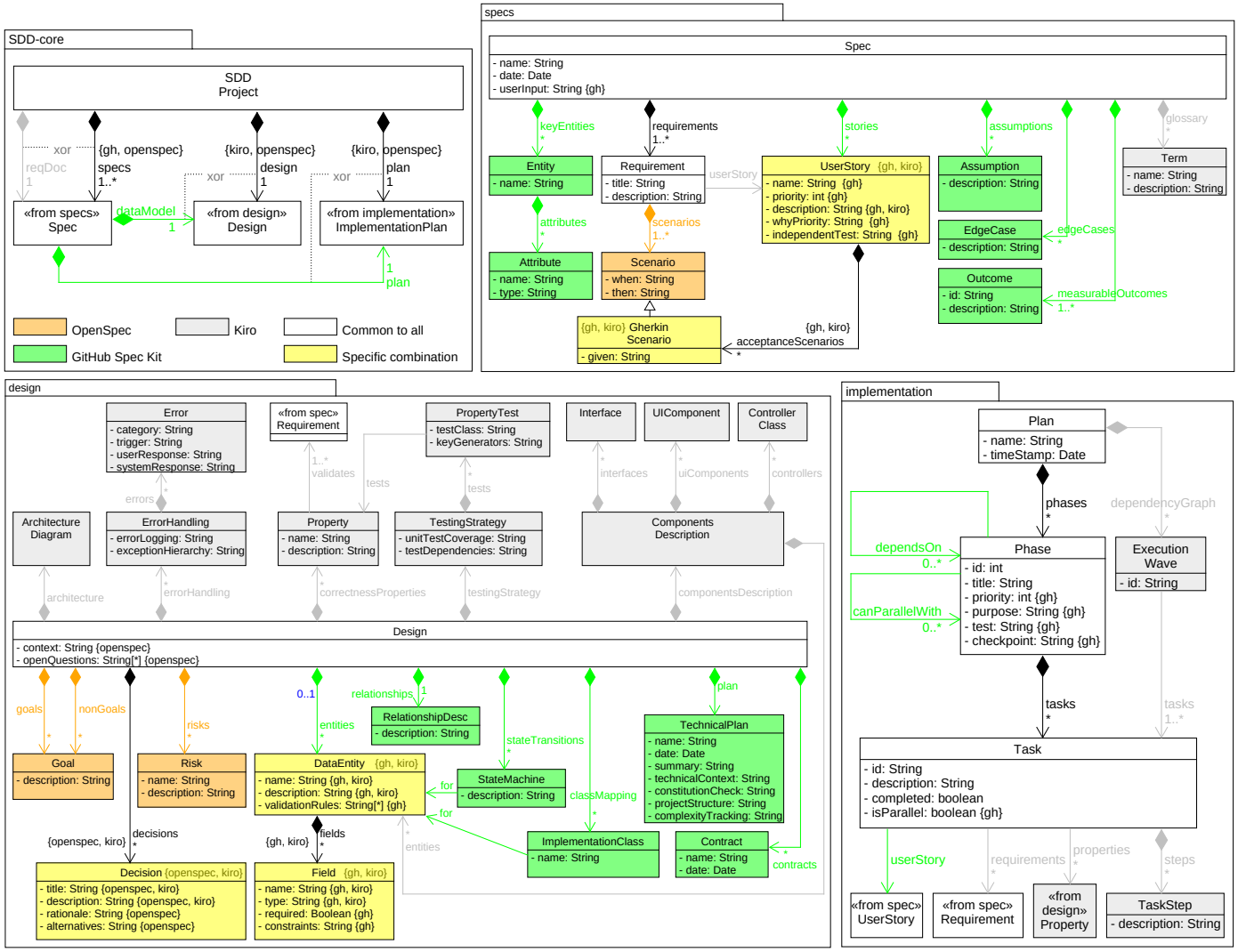

Figure 2 SDD metamodel based on GitHub Spec Kit, Kiro, and OpenSpec

At the requirements stage (package *specs*), all approaches rely on requirements lists (`Requirement`). Each of these requirements includes *WHEN/THEN* scenarios in OpenSpec (`Scenario`). Instead, Kiro and GitHub Spec Kit separate them into user stories, with Gherkin scenarios (*GIVEN/WHEN/THEN*) as used in behaviour-driven development (Adzic 2011). In addition, GitHub Spec Kit produces descriptions of domain entities (`Entity` with `Attributes`), assumptions, outcomes, and edge cases. Instead of a domain model, Kiro includes a glossary with the domain vocabulary.

The *design* package shows that specification mechanisms for this phase are more varied. The simplest approach is the one by OpenSpec, which only provides contextual descriptions, (non-)goal descriptions, risks, and open questions. Instead, the other SDD approaches embed models at this stage: state machines, a data model (`DataEntity` with `Fields`, and `RelationshipDesc`), architecture diagrams (in *Mermaid* format in the Markdown), description of components (interfaces, UI elements, controller classes), properties and contracts, and test plans (`TestingStrategy`). GitHub Spec Kit also includes a low-level design mapping from data entities to implementation classes (e.g., in Java).

In the *implementation* package, all approaches use a list of tasks organised in phases. Kiro includes more detailed steps, and traces each `Task` to the related `Requirements` and `properties`. Instead, GitHub Spec Kit traces `Tasks` to user stories. Both approaches provide additional information on how to execute the plan, organising parallelisable tasks into `ExecutionWaves` (Kiro), or providing explicit dependencies and parallelisation relations (GitHub Spec Kit).

As a concrete example, Table 2, summarises the specification and code size generated by each SDD approach, when building a simple application to handle cooking recipes. The initial prompt and the design instructions used in the following phase are indicated at the bottom of the table. For specification and code generation, we used GitHub Spec Kit, Kiro, and OpenSpec atop Claude Code CLI, using Sonnet (and/or Opus for Kiro) as the underlying LLM.

As hinted by the metamodel of Figure 2, OpenSpec is a much lighter SDD approach than the others. This is noticeable

Table 2 Specification and code size generated by each SDD approach on a sample project

Markdown Files	GitHub Spec Kit	Kiro	OpenSpec
Number of Files	8	3	10
Number of Lines	1,027	869	450
Number of Words	6,998	6,248	3,869
Generated LOC	1,032	3,754	1,162

Initial prompt: I am building an application to create and manage cooking recipes. The application should list recipes, organised by different tags, like vegetarian, or vegan; but the tags should be extensible. Each recipe has a name, description, ingredients, number of people and instructions. The application should allow flexible search for recipes, create new recipes, and enable annotations of the recipe. The application is mono-user, and recipes are stored locally. **Design instructions:** The application should be Java/Swing with as few dependencies as possible. No databases, but persistence should be in JSON. The application should be created in a standard Eclipse project.

in its fewer metamodel elements and in its smaller Markdown file size. On the other hand, Kiro and GitHub Spec Kit generate much larger specifications. Kiro generates much more code, and this is due to the fact that it is the only SDD approach that generates testing code (as hinted also by the detailed `TestingStrategy` in the metamodel). From the 3,754 LOC, 1,648 are for testing. Finally, of the three applications, the one generated by OpenSpec had problems persisting recipes, while the other applications worked with no problems. The code, specs, and extracted data of the projects are available at https://github.com/jdelara/SDD_MDE/.

With this conceptual model now in place, MDE practices can be realized on top of it, such as separation of concerns, abstraction engineering, collaboration and interoperability, support for analysis frameworks, and design space exploration (see Figure 1). This vision requires translating the natural language text found in specs to models and back. For this purpose, we can profit from works on bidirectional transformation approaches (Hidaka et al. 2016), and on blended modeling approaches (Ciccozzi et al. 2019). While specs in Markdown provides some structured support – especially when models are explicitly embedded in fenced blocks – more powerful translation approaches (probably based on AI) should be investigated in future work.

3.2. Embedding Each Other's Best Practices

The aforementioned bridge can be considered a pragmatic, operational interoperability solution. But there is not yet cross-fertilization that could improve either of the two independent paths.

Given that SDD is a recent development and is still evolving in response to the needs of the SE community and expertise of the AI community, it could benefit from the broader experience and lessons learned by the MDE community over many years. In particular, we highlight the following practices that could extend current SDD practices:

– *Separation of Concerns*. Right now, SDD often generates single Markdown files. While it is true that LLM contexts allow direct ingestion of large files, long context windows still represent a challenge. To aid human understanding, it would be better for agents to create separate files for each abstraction perspective, with a single index file that redirects the agent to the appropriate file based on the task at hand. Please note that OpenSpec divides requirements into several files, based on system features, rather than according to the principle of separation of concerns.
– A *refinement* of the abstractions to be made explicit in SDD files to enable greater analysis and exploration capabilities. SDD files tend to cover very well goal decomposition as well as structural and architectural aspects, but, for instance, behavioral information is often described in a much more fuzzy way. Only GitHub Spec Kit includes state machines, which is a step in the right direction.
– *Model-based analysis and verification*. Representing SDD artefacts as proper models could enable the reuse of established model-based analysis and verification methods and tools (Cederbladh et al. 2024; González & Cabot 2014; Heinrich et al. 2021), potentially strengthening reliability assurance in agentic systems. These tools could be exposed to agentic systems as specialised *skills*. This vision aligns with recent calls in software engineering for integrating AI-based development with formal methods (Meyer 2026).

Similarly, MDE can also learn from SDD. The fact that some abstractions are very present in SDD and not so much in MDE (e.g., modeling what the software is NOT about, such as OpenSpec's non-goals) should also be taken into consideration by the MDE community as these are aspects that we may want to integrate into standard MDE practices, especially in a context where the cost of producing some of these models could also be diminished thanks to the collaboration of the agents themselves. Note that modeling effort has always been linked to the return of investment of such models (i.e., how much benefit one can get by leveraging them later on in the development process, e.g., as the input to automatic code-generation processes or sophisticated analysis frameworks). If the cost of modeling would indeed decrease (yet another aspect that requires empirical validation; see Section 4.2), we can afford to model more concerns, especially those that agents in SDD consider valuable.

3.3. One Single Platform for Everybody

The previous steps enable communication between the SDD and MDE paths and enrich each with lessons learned from the other. But they are still two parallel paths that synchronize periodically.

A real integration would require real-time collaboration in a single live development platform able to mix all kinds of concrete notations to show to each type of user the best representation depending on the user profile and the abstraction at hand. Live and exploratory modeling editors (Döderlein et al. 2026) have been tried before, and projectional editors (Voelter et al. 2014) have shown the ability to mix different notation styles over a common metamodel. Coming from the AI / data science field, Jupyter-like interactive notebooks (LF charities 2015) have shown the usefulness and practicality of combining textual descriptions with executable and dynamic cells. A new

type of modeling editor that combines all these ideas and targets both humans and agents would be a logical next step in this integration. Such development platforms will blend free-form and structured Markdown-based specifications with live graphical modeling editors, allowing human developers and autonomous agents, orchestrated through Multi-Agent System Frameworks (MASFs) (Di Ruscio et al. 2026), to interact seamlessly over shared, synchronized representations. To this end, existing SDD approaches can also be exploited.

4. Future Plans

To realize the vision of integrating MDE with SDD, in this section, we outline our future plans in the form of a research roadmap structured around three main pillars: tooling implementation, empirical validation, and addressing practical adoption barriers.

4.1. Tooling Implementation

The main technical step is establishing the interoperability bridge. This requires explicitly defining the SDD metamodel for the core concepts embedded within SDD artifacts (e.g., requirements, data models, task plans). Formalizing the metamodel will enable the application of existing modeling tools, such as model transformations and model visualizations, in the context of SDD. The SDD metamodel presented earlier is a first step but must be validated against further SDD approaches based on a systematic review of existing SDD approaches. Furthermore, we plan to define a more detailed metamodel that also takes into account more completely an agent's harness including memory, skills, and protocols to provide the foundation for increased interoperability among SDD approaches. Each approach has its own way of defining system-level prompts, setting up skills, etc., and MDE can provide a tool-agnostic definition that enables transferring development between different coding agents, with direct practical relevance. This work can build upon existing initiatives, such as the Open Knowledge Format (McVeety & Hormati 2026) proposed by Google to standardize the structure of agent-managed knowledge bases. Additionally, tool interoperability can be realized through a unique metamodel architecture consisting of core and shared elements, and distinct extensions for tool-specific features, utilizing standard MDE operations like metamodel composition and model weaving.

Following the interoperability bridge effort, we envision the emergence of Jupyter-like integrated development environments (IDEs). However, developing novel IDEs requires a better understanding of human / agent collaboration and which kind of artifacts need to be shared, which needs to be investigated.

4.2. Empirical Evaluation

To demonstrate the cross-fertilization benefits of MDE and SDD, empirical validations are needed to assess the impact of the integration of SDD and MDE. First, the combined use of SDD and MDE in practice needs to be investigated. The metamodel presented in Section 3 is derived from *.md* files generated by different approaches throughout the SDD process, as well as from the canonical *.md* templates provided by the developers of the SDD approaches. While these artifacts offer valuable insights into the structure of specifications produced within SDD workflows, they may not fully reflect how specifications are created and maintained in real-world settings. To obtain a more representative understanding of SDD practices, we plan to mine open-source repositories for specification artifacts, analyse their content, and identify the model-like constructs and abstractions they contain. Existing corpus of spec-driven artefacts, such as (Agarwal et al. 2026), are also an excellent starting point for performing these additional analyses and validating the generalization of our results.

Furthermore, it is important to compare the performance of agents supporting software engineering tasks from raw natural language versus agents operating on a structured, MDE-generated model layer. Finally, it will also be strategic to evaluate how embedding explicit metamodel instances within an agent's memory bank (e.g., a Kiro steering file) impacts the number of iterations, the computational effort required to reach a verified solution, and the human actors' ability to understand and reason about the system.

4.3. Adoption Barriers

Several adoption barriers must be mitigated to transition this vision into mainstream practice. In particular, the definition of SDD and its intermediate artifacts is rapidly evolving, with many SDD approaches currently being available. Moreover, current coding agents obscure their internal decision-making processes (Liu et al. 2026). While injecting MDE practices aims to bring structural transparency and traceability, the inherent black-box nature of the underlying LLMs remains a challenge for verifiable software engineering. The lack of standardization across key enabling technologies for novel IDEs such as different MASFs further complicates the creation of IDEs.

Further concerns include the lack of large-scale benchmarks and the high API costs of conducting large-scale empirical studies with state-of-the-art coding agents, both of which limit the scope of academic experimentation. Additionally, deploying agentic workflows in critical domains requires overcoming strict legal and safety constraints, necessitating robust plans and formal verification steps that MDE is uniquely positioned to provide.

Overall, the proposed roadmap is inherently iterative as the field and its toolset continue to evolve rapidly. Thus, regular roadmap refinement, grounded in empirical feedback, will be essential to ensuring the sustained relevance and eventual success of this cross-fertilization effort.

Acknowledgements

Juan's work has been funded by the Spanish MICIU with project PID2024-155231OB-I00. Jordi's work is supported by the Luxembourg National Research Fund (FNR) through the PEARL program under grant agreement 16544475.

References

Adzic, G. (2011). *Specification by example: How successful teams deliver the right software*. Manning.

Agarwal, S., Singhal, A., Breaux, T., & Vasilescu, B. (2026). *Specmine: A large-scale corpus of spec-driven development artifacts.* Retrieved from https://arxiv.org/abs/2608.25202

Almonte, L., Rengifo, J. I., Guerra, E., & de Lara, J. (2026). Emf-kaizen: an intelligent assistant for domain-specific modelling and meta-modelling. In *ECMFA* (Vol. 25, pp. 1–14). doi: 10.5381/jot.2026.25.3.a8

Amazon Web Services. (2024). *Amazon q developer: Generative ai-powered assistant for software development.* https://aws.amazon.com/q/developer/. (Accessed: June 2026)

Amazon Web Services. (2025). *Kiro.* Retrieved from https://kiro.dev/ (Accessed: June 2026)

Anthropic. (2024). *Claude code: Agentic ai for software development.* Retrieved from https://www.anthropic.com/product/claude-code (Accessed: June 2026)

Anysphere Inc. (2022). *Cursor.* Retrieved from https://cursor.com/ (Accessed: June 2026)

Apostolou, S. A., Bosch, J., & Olsson, H. H. (2026). *Agentic ai in industry: Adoption level and deployment barriers.* Retrieved from https://arxiv.org/abs/2605.14675

Base44. (2026). *Base44.* Retrieved from https://base44.com/ (Accessed: June 2026)

Bencomo, N., Cabot, J., Chechik, M., Cheng, B. H., Combemale, B., Zschaler, S., et al. (2024). Abstraction engineering. *arXiv preprint*. Retrieved from https://arxiv.org/abs/2408.14074

Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice, second edition*. Morgan & Claypool Publishers.

Cámara, J., Troya, J., Burgueño, L., & Vallecillo, A. (2023, May). On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml. *Softw. Syst. Model.*, *22*(3), 781–793. Retrieved from https://doi.org/10.1007/s10270-023-01105-5 doi: 10.1007/s10270-023-01105-5

Cederbladh, J., Cicchetti, A., & Suryadevara, J. (2024, March). Early validation and verification of system behaviour in model-based systems engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, *33*(3). doi: https://doi.org/10.1145/3631976

Chaaben, M. B., Burgueño, L., & Sahraoui, H. (2023). Towards using few-shot prompt learning for automating model completion. In *Proceedings of the 45th international conference on software engineering: New ideas and emerging results* (p. 7–12). IEEE Press. Retrieved from https://doi.org/10.1109/ICSE-NIER58687.2023.00008 doi: 10.1109/ICSE-NIER58687.2023.00008

Chen, B., Chen, K., Hassani, S., Yang, Y., Amyot, D., Lessard, L., … Varró, D. (2023). On the use of gpt-4 for creating goal models: An exploratory study. In *2023 ieee 31st international requirements engineering conference workshops (rew)* (p. 262-271). doi: 10.1109/REW57809.2023.00052

Chen, K., Yang, Y., Chen, B., Hernández López, J. A., Mussbacher, G., & Varró, D. (2023). Automated domain modeling with large language models: A comparative study. In *2023 acm/ieee 26th international conference on model driven engineering languages and systems (models)* (p. 162–172). IEEE Press. Retrieved from https://doi.org/10.1109/MODELS58315.2023.00037 doi: 10.1109/MODELS58315.2023.00037

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., … Zaremba, W. (2021). *Evaluating large language models trained on code.* Retrieved from https://arxiv.org/abs/2107.03374

Ciccozzi, F., Tichy, M., Vangheluwe, H., & Weyns, D. (2019). Blended modelling - what, why and how. In *22nd ACM/IEEE international conference on model driven engineering languages and systems companion, MODELS companion 2019, munich, germany, september 15-20, 2019* (pp. 425–430). IEEE.

Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J. R., & Vojtisek, D. (2016). *Engineering Modeling Languages*. Chapman and Hall/CRC. Retrieved from https://inria.hal.science/hal-01355374

Contreras, A., Guerra, E., & de Lara, J. (2026, March). Architecting open, accountable, and trustworthy ai-ides. *Automated Software Eng.*, *33*(2). Retrieved from https://doi.org/10.1007/s10515-026-00608-x doi: 10.1007/s10515-026-00608-x

Delimarsky, D. (2025). *Spec-driven development with AI: Get started with a new open source toolkit.* GitHub. Retrieved from https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/ (Accessed: June 2026)

Di Ruscio, D., Nguyen, P. T., Di Sipio, C., Rubei, R., & Di Rocco, J. (2026). Engineering llm-based multi-agent systems: A taxonomy of emerging frameworks. *IEEE Software*, 1-8. doi: 10.1109/MS.2026.3694089

Döderlein, J.-B., Khelladi, D. E., Acher, M., & Combemale, B. (2026). *Spacetime programming: Live and omniscient exploration of code and execution.* Retrieved from https://arxiv.org/abs/2603.18735

Feng, S., Chen, B., Meyer, B. H., & Mussbacher, G. (2026). Llm-assisted repository-level generation with structured spec-driven engineering. In *Proceedings of the 34th acm international conference on the foundations of software engineering* (p. 1257–1261). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3803437.3805567

Fission AI. (2025). *Openspec.* Retrieved from https://github.com/Fission-AI/OpenSpec (Accessed: June 2026)

GitHub. (2021). *Github copilot: Your ai pair programmer.* Retrieved from https://github.com/features/copilot (Accessed: June 2026)

GitHub. (2025). *Github spec kit.* Retrieved from https://github.com/github/spec-kit (Accessed: June 2026)

González, C. A., & Cabot, J. (2014). Formal verification of static software models in mde: A systematic review. *Information and Software Technology*, *56*(8), 821-838. doi: https://doi.org/10.1016/j.infsof.2014.03.003

Heinrich, R., Durán, F., Talcott, C. L., & Zschaler, S. (Eds.). (2021). *Composing model-based analysis tools*. Springer. doi: https://doi.org/10.1007/978-3-030-81915-6

Hidaka, S., Tisi, M., Cabot, J., & Hu, Z. (2016). Feature-based classification of bidirectional transformation approaches.

Softw. Syst. Model., *15*(3), 907–928.

Jahan, M., Hassan, M. M., Golpayegani, R., Ranjbaran, G., Roy, C., Roy, B., & Schneider, K. (2024). Automated derivation of uml sequence diagrams from user stories: Unleashing the power of generative ai vs. a rule-based approach. In *Proceedings of the acm/ieee 27th international conference on model driven engineering languages and systems* (p. 138–148). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3640310.3674081 doi: 10.1145/3640310.3674081

Jellyfish. (2025). *Large-scale analysis of ai coding tool adoption and productivity impact across 1,000 companies.* Retrieved from https://www.zenml.io/llmops-database/large-scale-analysis-of-ai-coding-tool-adoption-and-productivity-impact-across-1-000-companies (Accessed: June 2026)

Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2026, January). A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, *35*(2). Retrieved from https://doi.org/10.1145/3747588 doi: 10.1145/3747588

Karpathy, A. (2025). *Vibe coding.* Retrieved from https://x.com/karpathy/status/1886192184808149383?lang=en (Accessed: June 2026)

LF charities. (2015). *Project jupyter.* Retrieved from https://jupyter.org/ (Accessed: June 2026)

Liu, J., Zhao, X., Shang, X., & Shen, Z. (2026). *Dive into claude code: The design space of today's and future ai agent systems.* Retrieved from https://arxiv.org/abs/2604.14228

Lovable. (2026). *Lovable: Ai-powered full-stack application development platform.* Retrieved from https://lovable.dev/ (Accessed: June 2026)

McVeety, S., & Hormati, A. (2026). *Introducing the open knowledge format.* Retrieved from https://cloud.google.com/blog/products/data-analytics/how-the-open-knowledge-format-can-improve-data-sharing

Mercado, J. S., Ma, Q., Cabot, J., Kelsen, P., & Proper, H. A. (2025). Towards human-in-the-loop llm-enabled domain modeling. In D. Bork, R. Lukyanenko, S. Sadiq, L. Bellatreche, & O. Pastor (Eds.), *Conceptual modeling - 44th international conference, ER 2025, poitiers, france, october 20-23, 2025, proceedings* (Vol. 16189, pp. 127–145). Springer. Retrieved from https://doi.org/10.1007/978-3-032-08623-5_7 doi: 10.1007/978-3-032-08623-5_7

Mermaid. (2014). *Mermaid editor and syntax.* Retrieved from https://mermaid.js.org/ (Accessed: June 2026)

Meske, C., Hermanns, T., Von der Weiden, E., Loser, K.-U., & Berger, T. (2025). Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access*, *13*, 213242-213259. doi: 10.1109/ACCESS.2025.3645466

Meyer, B. (2026). Artificial intelligence for software engineering: From probable to provable. *Communications of the ACM*, *69*(6), 46–49. Retrieved from https://dl.acm.org/doi/abs/10.1145/3773295 doi: 10.1145/3773295

Model Context Protocol Contributors. (2025). *Model context protocol specification.* https://modelcontextprotocol.io.

OpenAI. (2025). *Introducing codex: A cloud-based software engineering agent.* Retrieved from https://openai.com/codex (Accessed: June 2026)

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Software Eng.*, *50*(1), 85–105. Retrieved from https://doi.org/10.1109/TSE.2023.3334955 doi: 10.1109/TSE.2023.3334955

StackBlitz. (2026). *Bolt.new.* Retrieved from https://bolt.new/ (Accessed: June 2026)

Tabnine. (2026). *Tabnine - ai coding assistant.* Retrieved from https://www.tabnine.com/ (Accessed: June 2026)

Voelter, M., Siegmund, J., Berger, T., & Kolb, B. (2014). Towards user-friendly projectional editors. In B. Combemale, D. J. Pearce, O. Barais, & J. J. Vinju (Eds.), *Software language engineering* (pp. 41–61). Cham: Springer International Publishing. doi: 10.1007/978-3-319-11245-9_3

Wang, Y., Zhong, W., Huang, Y., Shi, E., Yang, M., Chen, J., … Zheng, Z. (2025, August). Agents in software engineering: survey, landscape, and vision. *Automated Software Eng.*, *32*(2). Retrieved from https://doi.org/10.1007/s10515-025-00544-2 doi: 10.1007/s10515-025-00544-2

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., & Cao, Y. (2023). React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations.* Retrieved from https://openreview.net/forum?id=WE_vluYUL-X

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., … Aftandilian, E. (2024, February). Measuring github copilot's impact on productivity. *Commun. ACM*, *67*(3), 54–63. Retrieved from https://doi.org/10.1145/3633453 doi: 10.1145/3633453