A Lean and Spec-Driven AI-Assisted Software Development Lifecycle for Applied AI Education: The AI-SDLC Approach

Andreas Martin and Sandro Schwander

FHNW University of Applied Sciences and Arts Northwestern Switzerland,
School of Business, Riggenbachstrasse 16, 4600 Olten, Switzerland
andreas.martin@fhnw.ch, sandro.schwander@fhnw.ch

Abstract. AI coding agents increasingly support software development beyond code completion, including planning, implementation, testing, and repository-level task execution. Their practical use, however, often remains only weakly connected to established software engineering practices. The aim of this work is to develop and evaluate a lightweight, spec-driven lifecycle for governed agentic software engineering. The lifecycle combines established software engineering practices with repository-local guidance through specifications, AGENTS.md, and phase-specific agent skill files. The approach was developed in the context of the FHNW course AI-assisted Software Development and applied by students to business-oriented software use cases. Its educational and practical applicability is explored through a student survey combining closed rating items with open-ended questions. The contribution of this work is a process-oriented framework that enables AI coding agents to operate with bounded autonomy within an explicit, reviewable, and test-oriented software development lifecycle.

Keywords: AI-assisted software development; agentic coding; spec-driven development; test-driven development; software development lifecycle.

1 Introduction

AI coding agents have become part of everyday software development. Generating code is no longer their most distinctive capability. Contemporary systems can interpret requirements, propose implementation strategies, generate tests, navigate repositories, and modify existing software artifacts [1], [2], [3]. As a result, the role of the developer is beginning to change.

The practical implications of this shift remain only partially understood. Much of the current literature concentrates on implementation performance, benchmark results, and coding productivity [3], [4]. Software development, however, involves more than implementation. Requirements must remain traceable, architectural decisions must remain coherent over time, and software artifacts must pass validation and deployment processes before they can be considered operational. Several authors have observed that current research and evaluation practices place considerably greater emphasis on implementation and testing than on requirements engineering, software design, deployment, or lifecycle governance [4]. Empirical studies of agentic coding point in a similar direction. AI-generated contributions can support development activities effectively, yet human review remains essential for quality assurance and contextual adaptation [5], [6].

Similar observations emerged during the development of the FHNW course AI-assisted Software Development. Students were generally able to produce functional software with

modern coding agents after a relatively short familiarization period. Difficulties appeared elsewhere. As projects evolved, requirements, architectural decisions, tests, and implementation artifacts increasingly drifted apart. Development activities often converged toward prompt-driven implementation, while the underlying rationale behind development decisions remained implicit and difficult to reconstruct.

The AI-Assisted Software Development Life Cycle (AI-SDLC) originated from this observation. The initial objective was not to develop another software engineering methodology. A more practical concern motivated the work. Students needed a lightweight process structure that would allow them to use AI coding agents productively without abandoning the engineering practices required for maintainable, testable, and deployable software systems.

The relevance of this problem extends beyond educational settings. Security-oriented studies suggest that apparently plausible and functionally correct agent-generated code may still contain vulnerabilities and quality deficiencies that are difficult to detect without systematic validation procedures [7]. At the same time, software teams increasingly experiment with agentic workflows that extend beyond isolated coding tasks. Under these conditions, productivity gains alone provide an insufficient basis for evaluation. The relationship between specifications, architecture, testing, deployment, and human oversight becomes equally important.

The purpose of this study is to develop and explore the applicability of the AI-SDLC as a lightweight lifecycle for more disciplined AI-assisted software development. The approach was applied in the FHNW course AI-assisted Software Development, where student teams used the lifecycle while developing business-oriented software solutions with contemporary coding agents. Following project completion, participants completed a short cross-sectional survey addressing process support, usability of the lifecycle artifacts, perceived usefulness of repository-local self-instruction, and the interaction between AI-generated outputs and human review activities.

2 AI-Assisted Software Development Lifecycle (AI-SDLC)

AI-SDLC represents one response to the challenges outlined above. The lightweight approach combines established software engineering practices, including Test-Driven Development (TDD) [8], the testing pyramid [9], Clean Architecture [10], CI/CD with continuous integration [11], containerized validation and continuous deployment with DevOps [12,13], Kanban-based execution [14], and selected concepts from the Scaled Agile Framework (SAFe) [15,16], within a lightweight lifecycle structure, together with more recent approaches to spec-driven development [17,18,19].

As depicted in Fig. 1, the activities are organized into six phases: Bootstrap, Specify, Design, Develop, Validate, and Deploy. These phases are intentionally minimal. Their purpose is to provide coordination, traceability, and lifecycle awareness rather than procedural rigidity. The AI-SDLC artifacts described in this section are available as a versioned software artifact on Zenodo [20].

Instead of relying exclusively on conversational prompts, AI-SDLC embeds development instructions directly into project artifacts. The `AGENTS.md` file [21] provides lifecycle guidance and operational constraints for coding agents. Additional `SKILL.md` files [22] define phase-specific behavior, while specifications, task-state documents, and architecture artifacts provide contextual grounding. Together, these elements establish a form of repository-local self-instruction that remains visible, reviewable, and reusable throughout the project lifecycle.

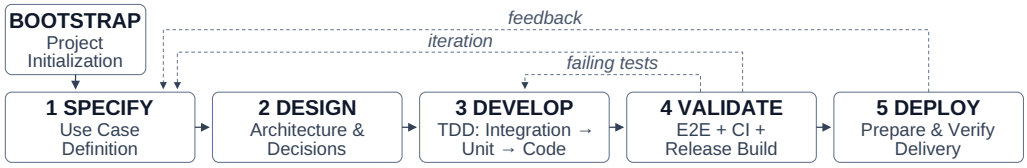


Fig. 1. Phases of the AI-Assisted Software Development Life Cycle (AI-SDLC)

2.1 Lifecycle Phases

The lifecycle starts with *Bootstrap*, which establishes the project context, development environment, and architectural baseline. Relevant information about the system, technology stack, architecture, and dependencies is maintained in `docs/PROJECT.md`. *Specify* then transforms a user story or change request into an explicit use case specification under `docs/specs/`, including intended behavior, scope, acceptance criteria, non-functional requirements, and intended tests. No implementation is performed during this phase.

During *Design*, the specification is mapped to the architecture following Clean Architecture. Required domain and application behavior, interfaces, infrastructure components, and dependencies are identified, consequential decisions can be documented as Architecture Decision Records (ADRs), and the implementation work is divided into tasks in `docs/TASKS.md`. *Develop* implements these tasks according to TDD, starting with integration and unit tests before producing the corresponding implementation through the Red–Green–Refactor cycle.

Validate considers the complete use case through automated unit, integration, and, where appropriate, end-to-end tests, together with CI and validation of the release artifact. Failed validation can return development to an earlier phase. Finally, *Deploy* prepares and verifies delivery of the validated artifact and, where authorized and configured, executes deployment to the target environment. Operational feedback or changed requirements can initiate another lifecycle iteration. AI-SDLC is therefore iterative rather than a linear or waterfall process.

2.2 Repository-Local Guidance for AI Coding Agents

A central characteristic of AI-SDLC is that development guidance and project context are maintained as version-controlled repository artifacts. Repository-level instruction files such as `AGENTS.md` provide an open format for supplying coding agents with persistent project-specific instructions [21]. Gloaguen et al. [23] found that coding agents generally follow such instructions, while larger context files do not necessarily improve task success and can increase inference costs.

Accordingly, AI-SDLC keeps `AGENTS.md` compact and uses it as the central entry point containing common workflow rules and guardrails. The current lifecycle state and tasks are maintained in `docs/TASKS.md`, architectural and runtime context in `docs/PROJECT.md`, and intended behavior in use case specifications under `docs/specs/`. Detailed procedural instructions are separated into phase-specific Agent Skills [22], each represented by a `SKILL.md` file under `skills/ai-sdlc-*`.

Consequential architectural decisions can additionally be recorded as Architecture Decision Records (ADRs), preserving their context, decision, and consequences [24]. Together, these artifacts separate general process guidance, phase-specific procedures, project context, lifecycle state, specifications, and architectural decisions. They persist across agent sessions and can be inspected, modified, and versioned together with source code and tests.

2.3 Human Control and Agent Autonomy

AI-SDLC allows coding agents to operate with substantial autonomy within individual lifecycle phases. Once a phase has been initiated and the required input artifacts are available, the agent can perform the activities defined by the corresponding skill, use available development tools, and create or modify the required artifacts without requiring human approval after every individual action.

Human control is maintained through the persistent artifacts connecting the lifecycle phases. Specifications, architectural decisions, tasks, tests, implementation artifacts, and validation results remain inspectable and can be modified whenever they do not adequately represent the intended requirements, architecture, or quality expectations. Modified artifacts subsequently become part of the agent's context for further activities. AI-SDLC thereby combines autonomous execution within lifecycle phases with persistent and reviewable artifacts through which developers retain control over the evolving software system.

3 Evaluation Method

We conducted a cross-sectional online survey among the students of a single course cohort. The questionnaire combined closed rating items with open-ended questions, so that the ratings could be read together with the situations respondents described in their own words. Since the study covers one cohort at one point in time and has no control group, it is descriptive and exploratory and is not designed to establish causal effects.

Participants were students of the bachelor's program Business Artificial Intelligence who had taken the module "AI-Assisted Software Development" and had carried out a semester project with the AI-SDLC. The version of AI-SDLC evaluated in the module is archived separately on Zenodo [25]. All students of the module were invited to take part. The invitation was issued after the module had been completed and graded, so that participation could not affect assessment. Participation was voluntary, and the survey link was not personalised, so responses cannot be traced back to individual students. The questionnaire was implemented in German. Of the 34 invited students, 23 opened the questionnaire and 13 submitted it completely, which corresponds to a response rate of 38% and, among those who started, a completion rate of 57%. Completeness was determined from the platform's disposition codes. The median time to complete the questionnaire was about 22 minutes.

The respondents' median age was 25 years (range 23–32). Programming experience was limited: five respondents described themselves as beginners and seven as having basic knowledge, with a median of three years of programming including studies and hobby projects, and prior development work had taken place mainly in hobby or open-source projects (10 of 13). Experience with AI coding tools, in contrast, was high: seven respondents had used such tools daily or intensively before the module, and none reported no experience at all.

3.1 Survey Structure

The questionnaire comprised six sections (A–F). Sections A and B collected participant background, prior software development experience, use of AI coding tools, and familiarity with software engineering concepts. Section C addressed the AI-SDLC in five thematic blocks covering process support, lifecycle artifacts, repository-local instructions, human

Table 1. Structure of the survey instrument

Part	Content
A) General questions	Age, degree program, current semester, experience
B) Previous experience	Contexts, process models, AI tools, familiarity with terms
C-I) Process support	Statements on the AI-SDLC as a whole
C-II) Lifecycle artifacts	Statements on user stories, architecture, specifications
C-III) Self-instruction files	Statements on AGENTS.md, skills, project-local rules
C-IV) Human verification	Statements on review, TDD and traceability
C-V) Learning and productivity	Statements on learning, speed and reuse
D) Phases	Adherence and perceived usefulness per phase
E) General feedback	Benefits, review practice, suggestions, remarks
F) Recommendation	Net Promoter Score item, consent for follow-up

verification, and learning and productivity. Statements were rated on a five-point agreement scale with an additional "not applicable / don't know" option. Each block contained at least one reverse-worded item to counteract acquiescence bias [26], and blocks C-I to C-IV included open-ended questions asking respondents to describe concrete situations from their projects.

Section D separately measured adherence to and perceived usefulness of each lifecycle phase, following the distinction between adherence and perceived value [27,28]. Section E contained four open-ended questions on benefits, review practices, improvements, and further remarks. Section F asked respondents how likely they would be to recommend AI-SDLC to a fellow student on a scale from 0 to 10 [29].

3.2 Analysis

All closed items were analyzed descriptively. Given the ordinal scales and the small sample, we report frequency distributions and absolute counts and use no inferential statistics. Reverse-worded items were retained in their original direction, and no composite scores were formed. Open-ended responses were coded thematically by one author; categories may overlap.

4 Findings

Overall, the AI-SDLC received broad approval (Fig. 2a). The clearest result concerns the Specify step: all 13 respondents agreed that it led them to sharpen the requirements before the AI started implementing, 11 of them strongly. Twelve of 13 agreed that the process helped structure the collaboration with the AI agent, and the same number that the clear sequence of phases gave them orientation in the project. Ten of 13 stated that the structured process improved the quality of their project result compared to their usual way of working. The open answers name goal-oriented feature design, clear guidelines and the step-by-step approach as the reasons; one respondent reported that the strict separation of Design and Validate prevented the agent from producing faulty code during a final refactoring. The counterweight is effort: four respondents agreed that the AI-SDLC felt like unnecessary overhead in individual phases, the free-text answers locate this criticism specifically in small tasks, where several respondents found the process disproportionate to the change being made.

Repository-local instruction files were judged positively throughout. Eleven of 13 respondents found it clear how to use and adapt such files, nine of them strongly. The reverse-worded control item points the same way: not a single respondent agreed that the files had hardly any noticeable influence, and 11 of 13 rejected the statement outright.

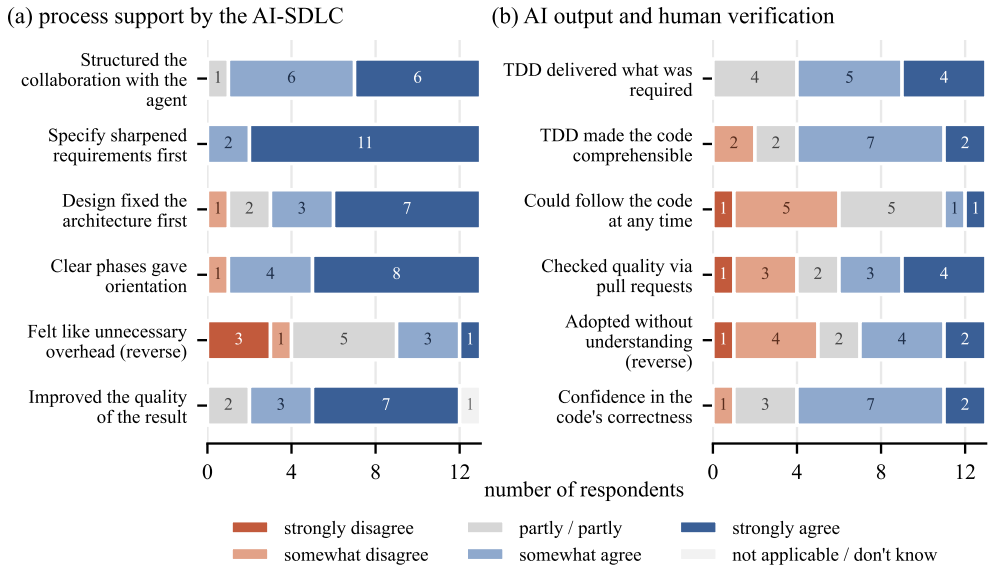


Fig. 2. Agreement with (a) the statements on process support by the AI-SDLC and (b) AI output and human verification ($n = 13$).

AGENTS.md was the most widely used file (10 of 13), followed by phase-specific skill files (5 of 13).

Human verification of AI output shows a more ambivalent picture (Fig. 2b). Respondents largely trusted the result: test-driven development kept the agent on what was actually required, and most were confident the code was correct, though mostly with reservations. This trust rested on the tests rather than on understanding the code. Following the agent's code at any time was one of the lowest-rated items in the survey, and only about half checked quality actively through pull requests. The free-text answers make this gap explicit: while six respondents admitted on the closed item to adopting suggestions without really understanding them, 11 of 13 described such a situation in their own words, citing fatigue in long sessions, time pressure, and features beyond their own programming skills. Verification thus shifted from reviewing code to trusting tests.

The same pressures shaped how the process itself was followed (Fig. 3). No phase was skipped, but only Develop was carried out fully by almost everyone, while Bootstrap, Specify and Validate were shortened most often (Fig. 3a). This was not a verdict on their value: every phase was rated useful, with means between 3.77 and 4.42 on the five-point scale (Fig. 3b). Eight of the nine respondents asked about it described shortening a phase, citing time pressure, effort, deliberate scope decisions for low-risk tasks, impatience at the start of the project, and in one case a lack of understanding. Bootstrap is the clearest case: as a one-time setup at the start, its effort comes before any visible benefit.

Asked about learning and productivity, all 13 respondents agreed that they made faster progress with the AI-SDLC, nine of them strongly, and 12 of 13 would use the process again voluntarily in a future project. Eleven of 12 agreed that it helped them understand software engineering concepts better. The one item that divides the group is self-directed learning: five respondents agreed that they had learned less themselves than they would have when programming everything on their own, four disagreed, and two were undecided.

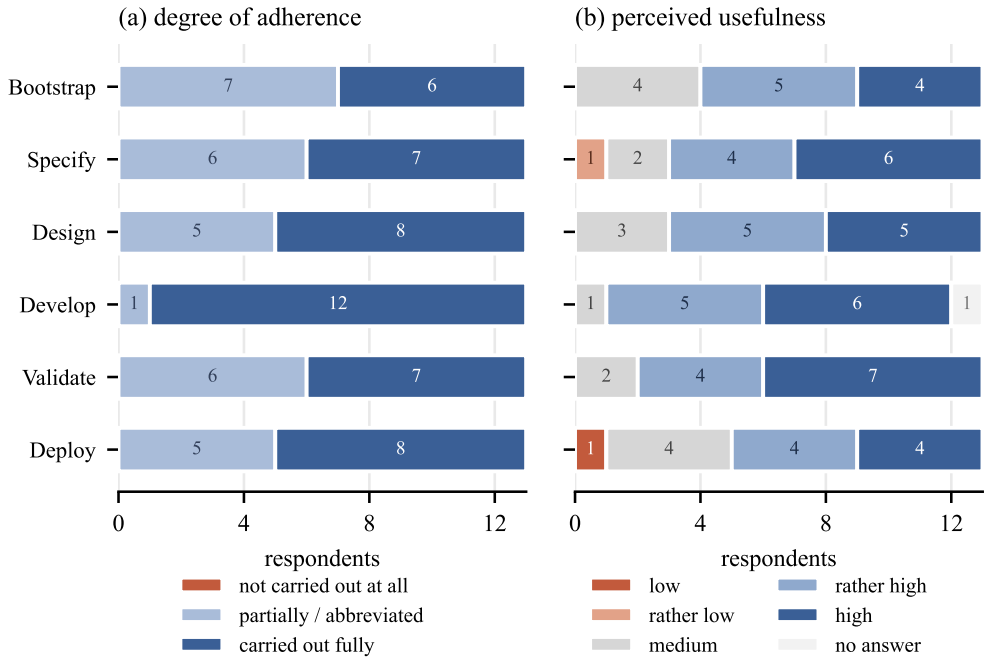


Fig. 3. Degree of adherence (a) and perceived usefulness (b) per lifecycle phase ($n = 13$), both in process order. No respondent reported skipping a phase entirely, which is why the level "not at all" appears in the legend of panel (a) without a corresponding bar.

Asked about the greatest benefit of the AI-SDLC, respondents clearly valued the structure and clear procedure most (8 of 12 substantive answers); early error detection in Specify and Design, confidence in code quality, and team collaboration followed far behind (3 mentions each). In checking AI-generated code, respondents focused on behaviour rather than on the code itself: trying the software out in operation was named most often (5 mentions), followed by automated tests and CI, while manual code reading was named no more often than cross-checking with a second agent or model (3 each). This fits the low rating for following the agent's code. Pull request review appears only once in the free text, although seven respondents had agreed on the closed item that they used it, suggesting it played a smaller role in practice than the ratings imply.

The suggestions for improvement fall into a small number of themes. Three respondents asked for a stronger or faster focus on tests, two for an explicit lightweight path for small tasks so that Specify and Design are not skipped ad hoc, and two for an additional step covering security and post-deployment monitoring. Individual respondents proposed specifying UI requirements as testable scenarios during the Design phase, reducing the initial effort of the Bootstrap phase through automation, and changes to team organisation. Two respondents would not change anything. Finally, the recommendation item shows a consistently positive, if not enthusiastic, picture: answers range from 7 to 10 with a median of 8, five respondents fall into the promoter range and none into the detractor range.

5 Conclusion and Future Work

The AI-SDLC was received positively by this cohort. Its clarity and the structure of the phases were the most frequently named benefit, and the repository-local instruction files were judged effective: 11 of 12 respondents agreed that they helped the agent adhere to project conventions, and not one respondent agreed that they had little noticeable effect. The rating items and the free-text accounts do not fully agree, however. Among the respondents who described a concrete case, more reported situations in which the instructions did not help than situations in which they did: conventions the agent forgot, an instruction file that grew large enough to consume too much context, and rules that had to be added only after a problem occurred. Repository-local instructions therefore appear to work, but neither automatically nor without maintenance.

The clearest open issue concerns the relationship between confidence and comprehension. Respondents reported confidence in the correctness of the generated code, yet the statement that they could follow and check that code at any time received the lowest agreement of any positively worded item in the survey, and 11 of 13 described a situation in which they had adopted code without fully understanding it. Verification also relied more on trying the software out in operation (5 mentions) than on automated tests and continuous integration (3 mentions). This points to two places where the process should be strengthened: the test structure before deployment, and an explicit step that supports understanding generated code rather than merely validating its functionality.

Three concrete improvements follow from the answers. First, artifacts and repository-local configuration files are not updated automatically. One respondent described in detail how this breaks down in teamwork: a change agreed in one developer's session is known to that developer's agent but not to the others', so decisions about implementation, architecture, and prioritization do not reach the whole team. Keeping specifications current is therefore a collaboration problem, not only a documentation one. This is particularly relevant to AI-SDLC, where agents can operate autonomously within lifecycle phases while developers retain control through reviewing and modifying persistent project artifacts. Second, architectural decisions were rarely recorded in structured form; ADRs were named by only one respondent. Although AI-SDLC provides ADRs for consequential architectural decisions, their role should therefore be made more explicit in practice. Third, one respondent had defined an own "new-feature" skill for a recurring task. Together with requests for a more lightweight process, this suggests a shorter path for small changes without requiring the full cycle.

Two limitations qualify these results. The cohort had little prior software engineering experience—five beginners and seven with basic knowledge, none very experienced—so most respondents had no structured process of their own against which to compare the AI-SDLC, and their assessment may be correspondingly favorable. With 13 respondents from a single cohort, the results describe this group and are not generalizable beyond it.

Future work follows directly from these points. The improvements outlined above should be implemented and the survey repeated with a later cohort, so that the changes can be assessed against a comparable measurement and the data basis broadened. A cohort with more professional software engineering experience would be particularly informative, since those participants could compare the AI-SDLC against a process they already know. Beyond that, measuring outcomes independently of self-report—code quality, test coverage, lifecycle adherence, or review activity in the project repositories—would address the most fundamental limitation of the present design, namely that the results reported here concern perceived rather than directly observed behavior.

Declaration on Generative AI Use. Generative AI assisted with language editing and Python implementation. Study design, analysis, interpretation, and verification remained with the authors.

References

1. Sapkota, R., Roumeliotis, K., Karkee, M.: Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI. ArXiv **abs/2505.19443** (May 2025)
2. Dong, Y., Jiang, X., Qian, J., Wang, T., Zhang, K., Jin, Z., Li, G.: A Survey on Code Generation with LLM-based Agents. ArXiv **abs/2508.00083** (July 2025)
3. Madani, M., Neumann, K., Nahhas, A., Chernigovskaya, M., Walia, D.S., Turowski, K.: Towards the Integration of Large Language Models into the Software Development Life Cycle: A Systematic Literature Review. In: 2025 3rd International Conference on Foundation and Large Language Models (FLLM). (November 2025) 773–781
4. Wang, K., Li, T., Zhang, X., Wang, C., Sun, W., Liu, Y., Shi, B.: Software Development Life Cycle Perspective: A Survey of Benchmarks for Code Large Language Models and Agents. ArXiv **abs/2505.05283** (May 2025)
5. Watanabe, M., Li, H., Kashiwa, Y., Reid, B., Iida, H., Hassan, A.: On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub. ACM Transactions on Software Engineering and Methodology (October 2026)
6. Abrahão, S., Grundy, J., Pezzè, M., Storey, M., Tamburri, D.: Software Engineering by and for Humans in an AI Era. ACM Transactions on Software Engineering and Methodology **34** (February 2025) 1–46
7. Zhao, S., Wang, D., Zhang, K., Luo, J., Li, Z., Li, L.: Is Vibe Coding Safe? Benchmarking Vulnerability of Agent-Generated Code in Real-World Tasks. ArXiv **abs/2512.03262** (December 2025)
8. Beck, K.: Test-Driven Development: By Example. Addison-Wesley, Boston, MA (2003)
9. Fowler, M.: Test Pyramid (May 2012) Published: martinfowler.com.
10. Martin, R.C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, Boston, MA (2017)
11. Fowler, M., Foemmel, M.: Continuous Integration (2006) Published: martinfowler.com.
12. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Signature Series (Fowler). Addison-Wesley, Boston, MA (2010)
13. Forsgren, N., Humble, J., Kim, G.: Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press, Portland, OR (2018)
14. Anderson, D.J.: Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press, Sequim, WA (2010)
15. Leffingwell, D.: Scaling Software Agility: Best Practices for Large Enterprises. Agile Software Development Series. Addison-Wesley, Boston, MA (2007)
16. Knaster, R., Leffingwell, D.: SAFe 5.0 Distilled: Achieving Business Agility with the Scaled Agile Framework. Addison-Wesley, Boston, MA (2020)
17. Piskala, D.B.: Spec-Driven Development: From Code to Contract in the Age of AI Coding Assistants. ArXiv **abs/2602.00180** (January 2026)
18. GitHub: Specification-Driven Development (SDD) (2025) Published: GitHub repository github/spec-kit.
19. Böckeler, B.: Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl (October 2025) Series: Exploring Gen AI Published: martinfowler.com.
20. Martin, A., Schwander, S.: The AI-SDLC Template and Method: AI-Assisted Software Development Life Cycle (September 2026) Version 1.0.0, Zenodo, DOI: 10.5281/ZENODO.22833065, URL: https://zenodo.org/doi/10.5281/zenodo.22833065.
21. AGENTS.md: AGENTS.md: A Simple, Open Format for Guiding Coding Agents (2026)

22. Agent Skills: Agent Skills: A Standardized Way to Give AI Agents New Capabilities and Expertise (2026)
23. Gloaguen, T., Mündler, N., Müller, M., Raychev, V., Vechev, M.: Evaluating AGENTS.md: Are Repository-Level Context Files Helpful for Coding Agents? (June 2026) arXiv:2602.11988 [cs.SE].
24. Nygard, M.: Documenting Architecture Decisions (November 2011)
25. Martin, A., Schwander, S.: The AI-SDLC Template and Method Pre-Release (June 2026) Version 1.0.0-beta, Zenodo, DOI: 10.5281/ZENODO.22724343, URL: https://zenodo.org/doi/10.5281/zenodo.22724343.
26. Weijters, B., Baumgartner, H.: Misresponse to reversed and negated items in surveys: A review. Journal of Marketing Research **49**(5) (2012) 737–747
27. Carroll, C., Patterson, M., Wood, S., Booth, A., Rick, J., Balain, S.: A conceptual framework for implementation fidelity. Implementation Science **2**(1) (2007) 40
28. Davis, F.D.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. MIS Quarterly **13**(3) (1989) 319–340
29. Reichheld, F.F.: The one number you need to grow. Harvard Business Review **81**(12) (2003) 46–54