

Leiden University

ICT in Business and the Public Sector

Can you specify the vibes? Designing and evaluating a Spec-Driven Development Tool for Data Products

Name:	Shanti Lorena Kuipers
Student ID:	s3258068

Date:	25/08/2026

1st supervisor:	Prof. dr. ir. J.M.W. Visser
2nd supervisor:	Dr. G.J. Ramackers
Company supervisor:	J. Schalken

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Einsteinweg 55
2333 CC Leiden
The Netherlands

Abstract

Background: Generative AI tools increasingly allow developers to create software by describing what they want in natural language. This has contributed to vibe coding, an informal development approach in which users repeatedly prompt AI tools to generate and change software, often without closely reviewing the generated code. Vibe coding can support rapid prototyping, productivity, and experimentation, but it can also lead to uneven code quality, limited quality assurance, security risks, maintainability problems, and over-reliance on AI. In data-product development, informal prompts can also leave important data-related requirements unclear, including data sources, transformations, metadata, infrastructure, data quality, and access requirements. Spec-Driven Development (SDD) may offer a more structured alternative by using written specifications to state requirements clearly and guide AI-assisted implementation.

Aim: This research investigates how a Spec-Driven Development tool can support efficient data-product development.

Method: We follow a Design Science Research approach. We review literature on Spec-Driven Development, software requirements specifications, and data products, and compare existing Spec-Driven Development tools. Based on this comparison, we select OpenSpec as the basis for the artefact. We extend it with skills that ask users questions to clarify their requirements, extend specifications, validate artefacts, configure the workflow, and provide domain-specific guidance. We first demonstrate the tool using a synthetic data-product case. We then evaluate it with four practitioners through individual think-aloud sessions and a Technology Acceptance Model questionnaire. Finally, two of these participants apply the refined tool to their own ongoing development tasks during a joint practical-evaluation session.

Results: The tool asks users to clarify their requirements and generates a structured proposal, specifications, a design, and implementation tasks before implementation. Participants value the clear documentation, consistent structure, and clarification questions. On a five-point scale, the mean scores for items measuring perceived usefulness range from 4.25 to 5.00, while those measuring perceived ease of use range from 4.25 to 4.75. The average willingness-to-use score is 4.25. The controlled evaluation shows that the clarification questions need to address data sources, attributes, join conditions, transformation logic, and data models more clearly. We refine the tool based on these observations. During the practical evaluation, both participants apply the refined workflow to ongoing development tasks and are able to progress through it.

Conclusion: The results show that a Spec-Driven Development tool can support data-product development by helping users create structured specifications before implementation. The proposed OpenSpec extension also supports clarification, domain-specific guidance, and validation. The tool does not replace developer judgement, but provides a structured starting point for development and documentation. Further research is needed to measure its effect on development time and rework in larger and longer-term evaluations.

Contents

Abstract . . . 1

1 Introduction **4**
1.1 Problem statement . . . 4
1.2 Objective . . . 5
1.3 Research questions . . . 5
1.4 Overview of the thesis . . . 5

2 Background and related work **6**
2.1 AI-assisted software development . . . 6
2.1.1 Types of AI-assisted development tools . . . 6
2.1.2 Vibe Coding . . . 7
2.2 What is Spec-Driven Development? . . . 9
2.2.1 The Spec in Spec-Driven Development . . . 13
2.3 Data-as-a-Product . . . 14
2.3.1 From Data Warehouses to Data Mesh . . . 15
2.3.2 What are data products? . . . 17
2.4 Case context . . . 20
2.4.1 Data product lifecycle . . . 20
2.4.2 Data catalog . . . 20
2.4.3 Data product development . . . 20

3 Method **22**
3.1 Design Science Research . . . 22
3.1.1 Problem identification and motivation . . . 23
3.1.2 Objectives of a solution . . . 23
3.1.3 Design and development . . . 23
3.1.4 Demonstration . . . 24
3.1.5 Evaluation . . . 24
3.1.6 Communication . . . 25
3.2 Research context and scope . . . 25
3.3 Data collection . . . 25
3.3.1 Design and demonstration data . . . 26
3.3.2 Controlled-evaluation data . . . 26
3.3.3 Practical-evaluation data . . . 26
3.3.4 Data handling . . . 26
3.4 Data analysis . . . 26
3.4.1 Analysis of design data . . . 26
3.4.2 Analysis of demonstration data . . . 27
3.4.3 Analysis of controlled-evaluation data . . . 27
3.4.4 Analysis of practical-evaluation data . . . 27

4 Design **28**
4.1 Analysis of existing SDD tools . . . 28
4.1.1 SDD tools . . . 28
4.1.2 Tool comparison . . . 29
4.1.3 Design insights . . . 33

4.2 Tool selection . . . 33
4.2.1 OpenSpec . . . 35
4.2.2 Refinement through Databricks-specific skill . . . 35
4.3 Software Requirements Specification analysis . . . 38
4.3.1 Mapping OpenSpec to SRS sections . . . 38
4.3.2 Mapping SRS sections to data product components . . . 41
4.4 Tool Design . . . 43
4.4.1 Extension skills . . . 44
4.4.2 Domain skills . . . 45
4.4.3 configure-openspec . . . 46
4.5 Tool Development . . . 46

5 Evaluation **50**
5.1 Demonstration . . . 50
5.1.1 Demonstration setup . . . 50
5.1.2 Observations and refinements . . . 50
5.1.3 Refined demonstration . . . 51
5.2 Controlled evaluation . . . 54
5.2.1 Controlled evaluation setup . . . 54
5.2.2 Session procedure . . . 54
5.2.3 Think-aloud observations . . . 55
5.2.4 Technology Acceptance Model . . . 57
5.2.5 Refinements . . . 57
5.3 Practical Evaluation . . . 57
5.3.1 Evaluation setup . . . 58
5.3.2 Practical evaluation observations . . . 58
5.3.3 Follow-up Technology Acceptance Model . . . 59

6 Discussion **60**
6.1 Overall interpretation of the evaluation findings . . . 60
6.2 Implications for Spec-Driven Development . . . 60
6.2.1 Specifications as driver for implementation . . . 60
6.2.2 SDD as a response to vibe coding . . . 61
6.2.3 The continuing role of the developer . . . 62
6.3 Applying SDD to data-product development . . . 63
6.3.1 Required specification content . . . 63
6.3.2 The need for contextual grounding . . . 64
6.3.3 Scope of data-product applicability . . . 64
6.4 Practical and design implications . . . 65
6.4.1 Documentation and traceability . . . 65
6.4.2 Learning curve, adoption, and complementary tools . . . 65
6.4.3 Design principles for SDD workflows . . . 66
6.5 Limitations . . . 66

7 Conclusion **68**
7.1 Answers to the research questions . . . 68
7.2 Contributions . . . 70
7.3 Future work . . . 70

A SDD Definitions **75**

B spec elements **77**

C Mapping Spec elements to IEEE 830 SRS **79**

D Data product technical components **81**

E Data product case **83**
E.1 Synthetic case . . . 83

E.2 Controlled-evaluation case 83

F DAB **84**

G TAM questionnaire **85**

H SDD tool analysis - generated files **86**

1 Introduction

This chapter introduces the research conducted in this thesis. It first presents the problem addressed and motivates the application of Spec-Driven Development to AI-assisted data-product development. It then defines the research objective and scope, formulates the main research question and supporting sub-questions, and concludes with an overview of the remaining chapters.

1.1 Problem statement

Generative AI is changing software development. It allows developers to describe intended functionality in natural language and let AI tools carry out increasingly large parts of the implementation [37, 42]. This has contributed to the emergence of vibe coding, an informal approach in which users repeatedly prompt an AI tool to generate and change software, often without closely reviewing the generated code [20, 24, 31]. Vibe coding can support rapid prototyping, productivity, and experimentation. Natural-language interaction can also make software development more accessible to people with limited programming experience [20]. However, when development is based mainly on informal prompts, requirements, assumptions, and design decisions may remain implicit. The literature identifies risks related to uneven code quality, limited quality assurance, security, maintainability, debugging, over-reliance on AI, and accountability [20]. Therefore, when AI tools take on more development work, developers need to provide clear and reviewable requirements that describe what should be built.

Spec-Driven Development (SDD) has emerged as a more structured approach to AI-assisted development. In SDD, specifications are the main development artefacts. They are created before implementation and used to guide the generation and verification of software [16, 39, 47, 55]. This retains natural-language interaction while making requirements and development decisions easier to review. However, SDD is a recent concept that is mainly described in practitioner and grey literature. It does not yet have a consistent formal definition, and limited academic research examines how its principles can be turned into concrete workflows and tools. It is also unclear how general-purpose SDD approaches should be adapted to contexts that require domain-, platform-, and organisation-specific information. Having a specification does not by itself ensure that an AI tool uses the relevant context.

These issues are also relevant to data-product development. Data products are managed artefacts that transform and package data to satisfy recurring information needs [26]. Developing them requires more than describing functional behaviour. Developers must consider data sources, selected attributes, relationships between datasets, transformation logic, and output structures. They must also account for metadata, infrastructure, data quality, security, and access requirements [8, 26, 30, 38, 53, 60]. These details are important for analytical or integrated data products because these products combine or expose data across domains to support cross-domain analysis. When the required information is not stated clearly, a programming agent may make unsupported assumptions about sources, attributes, join conditions, transformations, or technical choices. The resulting specifications and implementations may appear reasonable but not match the intended data product or development environment.

Within the case organisation, analytical or integrated data products are developed in Databricks and follow organisational development requirements. AI-assisted development is becoming relevant in this setting, but the organisation does not yet have an established specification-driven approach adapted to this context. Existing SDD tools provide general workflows for turning an initial request into specifications and implementation activities. However, it is not yet established how these workflows should include the technical components of data products, relevant specification categories, and knowledge about the platform and organisation.

The research problem is therefore the lack of an established and evaluated approach for applying SDD to the technical specification and AI-assisted development of analytical or integrated data products. Without such an approach, important specifications may remain unspecified or be replaced by unsupported interpretations from the programming agent. This also creates a research gap. It is not yet clear how SDD principles and established software-requirements structures can be combined with data-product

specifications and contextual knowledge. It is also unclear how such a tool can support clarification and validation while keeping developers responsible for reviewing the generated artefacts. To address this gap, we design and evaluate a Spec-Driven Development tool that provides workflow structure while keeping developers responsible for reviewing specifications and implementation outcomes.

1.2 Objective

The objective of this thesis is to design and evaluate a Spec-Driven Development tool to support AI-assisted data-product development. The tool is informed by principles of Spec-Driven Development (SDD), insights from existing Spec-Driven Development tools, and the characteristics of data-product development within a data mesh context. The study focuses on analytical or integrated data products and on the early stages of the data-product lifecycle, particularly the initiation and build phases. Activities related to publication, delivery, consumption, refinement, and decommissioning are considered outside the scope of the study. The design aims to support more structured AI-assisted data-product development. The proposed tool is evaluated through its application to a representative data-product development case, participant feedback on its usability, clarity, and perceived support for development work, and a practical evaluation in which the refined tool is applied to ongoing development tasks.

1.3 Research questions

Building on the problem statement, this thesis aims to design and evaluate a Spec-Driven Development tool that supports efficient data-product development. The main research question guiding this thesis is:

RQ How can a Spec-Driven Development tool be designed to support efficient data-product development?

To address the main research question, a set of sub-questions is defined. These sub-questions structure the research by guiding the conceptualization of Spec-Driven Development and data-product development, the analysis of existing tools, the derivation of design requirements, and the design and evaluation of the proposed artefact.

SQ.1 How can Spec-Driven Development be defined?

SQ.2 What are the characteristics of specifications in Spec-Driven Development?

SQ.3 What are the characteristics of data product development?

SQ.4 What design insights can be derived from existing Spec-Driven Development tools?

SQ.5 What requirements should a Spec-Driven Development tool satisfy for data product development?

SQ.6 How effective is the proposed tool in supporting data product development?

1.4 Overview of the thesis

In Chapter 2, we discuss the background and related work by covering three main literature streams: (1) AI-assisted software development and "vibe coding", (2) Spec-Driven Development and the characteristics of its specifications, and (3) data products and their development within data mesh architectures. Chapter 2 also presents the case context, describing the data-product lifecycle, development process, and tooling in the case organisation. Chapter 3 presents the research approach, Design Science Research, and explains how it guides the iterative design, development, demonstration, and evaluation of the artefact: a Spec-Driven Development tool for data-product development. Chapter 4 presents the design and development of the tool, including the analyses informing its design and the resulting workflow and components. Chapter 5 reports on the evaluation results, assessing the tool through a technical demonstration, a controlled evaluation with practitioners, and a subsequent practical evaluation using ongoing development tasks. Chapter 6 discusses what the design and evaluation results mean for applying Spec-Driven Development to data-product development and reflects on its practical and design implications, scope, and limitations. Finally, Chapter 7 concludes the thesis by answering the research questions, summarising the main contributions to research and practice, and outlining directions for future work.

2 Background and related work

This chapter establishes the theoretical and practical background required to understand the proposed artefact. It first introduces AI-assisted software development and distinguishes between conversational, IDE-integrated, and autonomous tools. It then examines vibe coding, including its potential benefits and limitations, as the context from which the need for a more structured development approach emerges.

The chapter subsequently introduces Spec-Driven Development, related software development approaches, its documented workflows and tools, and the role of specifications. The relationship between SDD specifications and the IEEE 830 Software Requirements Specification structure is also examined. The chapter then discusses data as a product, including relevant data architectures, the definition and lifecycle of data products, and their technical components. Finally, it describes the organisational case context and the environment in which data products are developed. Together, these topics provide the conceptual basis for the design and evaluation of the proposed tool.

2.1 AI-assisted software development

AI tools such as ChatGPT and Claude make interaction with Large Language Models (LLMs) widely accessible. People use these tools to summarise text, support writing, and ask questions in both everyday and professional contexts [44]. At the core of these tools are LLMs: machine-learning models trained on large amounts of text to predict the next word or token. This ability allows them to generate coherent responses and communicate with users through natural language [4].

Software development provides a clear example of how this interaction changes existing ways of working. Stack Overflow is a common source of answers when developers encounter complex programming problems. However, instead of searching through previously posted questions and answers, developers increasingly turn to LLM-based tools to describe their specific problem and receive an immediate, tailored response [64]. These tools can explain technical concepts, suggest solutions, and generate code that developers can subsequently review and adapt.

The role of these tools extends beyond answering questions or generating individual pieces of code. More autonomous tools take on larger parts of the development process by generating, modifying, and testing code with less direct human input [37]. In this thesis, these more autonomous tools are referred to as *programming agents*.

The increasing role of programming agents changes how developers communicate what they want a computer to do. Rather than only expressing their intentions through code written line by line, developers can describe the intended result in natural language and subsequently review and refine the generated output. This does not remove the need to understand software, but shifts part of the developer's effort from manually writing code towards specifying intended functionality and reviewing, validating, and integrating the generated output [6, 42].

Figure 2.1 places this change within the broader evolution of programming. The progression begins with machine and assembly code, in which instructions closely correspond with the computer's underlying language. It then moves towards high-level languages, structured and object-oriented programming, and reusable design patterns. Each stage introduces a higher level of abstraction, allowing developers to express their intentions in ways that are increasingly removed from the computer's raw instruction format. In the most recent stage, predictive tools and LLMs make natural-language prompts part of the programming workflow. Programming agents build on this development by using these instructions to perform larger parts of the development process [6, 42].

2.1.1 Types of AI-assisted development tools

AI-assisted development tools are described as three main types [10]:

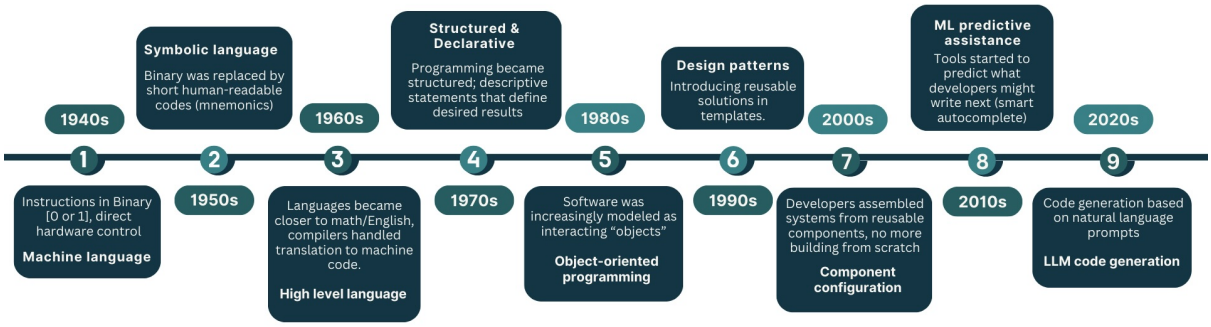


Figure 2.1: How we tell computers what we want: the evolution of programming (1940s–2020s) [6, 42]

1. **Conversational:** Developers describe a problem in text and receive explanations, example code, or step-by-step guidance. However, this includes manual code transfer and loses context between sessions.
2. **IDE-integrated:** The tool completes code directly within an IDE (Integrated Development Environment - program such as Visual Studio Code where developers can edit, run, and debug code). It is especially used for suggesting blocks of code, rewriting parts more cleanly, and helping the developer stay "in the flow" because it works in the same window with immediate feedback and the current code as context. However, it is usually less helpful for big, high-level design decisions about how an entire system should be structured.
3. **Autonomous agent:** The tool can make changes across multiple files based on high level instructions.

These approaches differ mainly in how much control stays with the developer versus the tool. Conversational and IDE (Integrated Development Environment) assistants typically support the developer's workflow, while autonomous agents aim to take over larger portions of the task [10].

Agent skills

Programming agents can be extended through skills. A skill is a reusable package of instructions and supporting resources that guides an agent in performing a particular type of task. Unlike an ordinary prompt, which provides instructions within a single interaction, a skill stores procedural knowledge that can be applied again when a relevant task arises. A skill typically consists of a directory containing a `SKILL.md` file with its name, description, and instructions, and may also include scripts, reference documents, or other resources. The programming agent initially uses the name and description to determine whether the skill is relevant and loads the complete instructions and supporting resources only when they are needed [62].

2.1.2 Vibe Coding

The most recent phase in Figure 2.1, LLM code generation, enabled the ability to create software by describing what is wanted in natural language and allowing an AI tool to generate the corresponding code. Within this context, a way of coding with AI emerged that is commonly referred to as *vibe coding*. The term was introduced by Andrej Karpathy in early 2025 [31], when he wrote:

> *"There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It's possible because the LLMs (e.g., Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so I barely even touch the keyboard. I ask for the dumbest things like 'decrease the padding on the sidebar by half' because I'm too lazy to find it. I 'Accept All' always, I don't read the diffs anymore. When I get error messages I just copy paste them in with no comment, usually that fixes it. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding—I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works."*

Karpathy's description presents vibe coding as a conversational and iterative process in which users describe the intended result, accept generated changes with limited review, and return errors to the AI tool. Based on this description and a review of practitioner literature, Fawzy et al. [20] define vibe coding as "the practice of using AI code generation tools to produce software primarily by describing goals in natural language and iteratively prompting, while relying on minimal review of the generated code." This limited engagement with the implementation was also observed by Geng et al. [24]. In their study, students mainly interacted with Replit to test and debug their prototypes and rarely inspected the generated code. Advanced software engineering students did, however, include more information about the intended functionality and existing codebase in their prompts than introductory students. Vibe coding can therefore be distinguished from broader AI-assisted programming, in which developers may use AI while continuing to plan, inspect, test, and manually modify the implementation [20].

Table 2.1 summarises the comparison between vibe coding and traditional coding presented by Bada et al. [6]. The comparison mainly concerns how users express their intentions, the technical knowledge required, and the extent to which they engage directly with the code.

Aspect	Vibe coding	Traditional coding
Syntax dependency	Low	High
Creativity	Supported through collaboration with AI	Constrained by formal programming rules
Accessibility	Accessible to a wider population, including non-programmers	Requires programming knowledge and skills
Interaction method	Conversational interaction using natural language	Directly writing code line by line
Focus	Shaping and refining an idea	Problem-solving guided by programming rules
Code understanding	Full understanding of the generated code is not necessarily required	The developer is expected to understand the code they author

Table 2.1: Differences between vibe coding and traditional coding, adapted from Bada et al. [6].

In Bada's comparison, the use of natural language reduces the dependence on programming syntax and makes software creation accessible to users with less programming experience. It also shifts the user's focus from directly implementing individual lines of code towards describing and refining the intended result. However, the reduced need to inspect the implementation is also related to several limitations identified in the literature. The following subsection therefore considers both the potential benefits and limitations of the approach.

Potential benefits and limitations

Different benefits and limitations of vibe coding have been identified in the literature. The reported benefits mainly relate to rapid prototyping, accessibility, productivity, experimentation, creativity, and collaboration. At the same time, the literature discusses limitations concerning code quality, quality assurance, security, maintainability, debugging, reliance on AI, and accountability. Table 2.2 provides an overview of the potential benefits, while Table 2.3 presents the potential limitations.

Benefit	Description
Rapid prototyping	Ideas can be converted into functional prototypes quickly [23, 20, 35].
Accessibility	Natural-language interaction allows people with limited programming experience to create software [23, 20, 35].
Productivity	Functionality can be generated and modified with less manual coding [23, 20, 35].
Experimentation and creativity	Ideas and alternative solutions can be generated and evaluated at relatively low cost [23, 20, 35].
Collaboration	People with different technical backgrounds can contribute to the development of prototypes [23, 35].

Table 2.2: Potential benefits of vibe coding identified in the literature.

Limitation	Description
Uneven code quality	Generated code may be incorrect, inconsistent, or require substantial rework [23, 20].
Limited quality assurance	Testing and code review may be skipped or delegated to the AI tool [20].
Security risks	Generated code may contain vulnerabilities that are not immediately visible [23, 20].
Maintainability problems	Limited understanding of generated code may contribute to technical debt and make later changes more difficult [20].
Debugging difficulties	Users may struggle to resolve problems when generated code becomes complex, buggy, or inconsistent [20, 35].
Premature convergence	Rapid generation may encourage users to accept the first workable solution without sufficiently considering alternatives [23, 35].
Over-reliance on AI	Dependence on generated solutions may reduce engagement with the code and technical learning [23, 35].
Limited accountability	Responsibility and the reasoning behind generated changes may become unclear [23, 35].

Table 2.3: Potential limitations of vibe coding identified in the literature.

Together, the identified benefits and limitations indicate a trade-off between rapid development and long-term software quality. Vibe coding can support the quick exploration and implementation of ideas, but a functional result does not necessarily mean that the underlying code is correct, secure, or maintainable [20, 23]. The relevance of these limitations may depend partly on the user's technical experience. For example, the advanced participants studied by Geng et al. [24] included more application and codebase context in their prompts, while [23] found that technical expertise supported the identification of problems, the selection of appropriate technologies, and the alignment of generated solutions with the intended goals.

To address the identified limitations, the literature discusses practices such as dividing large tasks into smaller steps, considering alternative solutions before selecting an approach, and applying testing, code review, quality gates, and traceable decision records [20, 23]. These practices add structure and validation to AI-assisted development. The need to retain the benefits of AI-assisted development while making requirements, decisions, and validation more explicit provides the transition to Spec-Driven Development, which is discussed in the following section.

2.2 What is Spec-Driven Development?

Although specification-oriented development approaches existed before generative AI, the modern AI-focused interpretation of Spec-Driven Development (SDD) gained attention in practitioner literature in 2025 [16, 39, 55]. Several sources present SDD as a more structured response to the benefits and limitations of vibe coding discussed in the preceding section. While vibe coding supports rapid prototyping, its limited quality assurance and accountability can make it difficult to verify whether generated code meets the intended requirements and to trace the decisions made during development [16, 20, 23, 39]. SDD aims to retain the speed and natural-language interaction of AI-assisted development while introducing persistent and reviewable specifications that document the intended system and provide a basis for implementation and verification.

Although the term Spec-Driven Development appears in practitioner literature, a consistent formal definition is lacking. Therefore, this study synthesizes existing descriptions from practitioner and grey literature to derive a working definition. The elements presented in Table 2.4 form the basis for this definition. These elements were identified through coding of the reviewed sources, a complete definitions table can be found in Appendix A.

Across the reviewed sources, specifications are consistently described as the primary artefact of development, indicating that they are central to the development process rather than supplementary documentation. This is closely connected to the spec-first principle, where development begins with the creation of specifications before implementation takes place. In addition, the sources emphasize that specifications describe what the system should do rather than how it should be implemented, focusing on intended behavior instead of technical realization. Beyond describing system behavior, specifications

Definition element	Characteristic	Source
primary artefact	Specifications are the primary artifact of development	[9, 16, 39, 45, 47]
spec-first	Development begins with writing specifications	[9, 47, 63]
what not how	Specifications describe what the system should do, not how	[9, 40, 55]
spec drives implementation	Specifications drive implementation	[47, 55]
AI generation and verification	AI agents generate and verify implementations based on specifications	[9, 45, 47, 55]

Table 2.4: Definition elements and characteristics of Spec-Driven Development based on reviewed literature; see Appendix A.

are also characterized as actively driving the implementation, meaning that they guide how the system is constructed. Finally, several sources highlight the role of AI-based generation and verification, where programming agents use the specification to generate and validate implementations.

Taken together, these elements indicate that specifications in Spec-Driven Development are not only descriptive artifacts but also play an active role in guiding and enabling implementation. Based on these recurring characteristics, the following working definition is proposed.

Definition 1 (Spec-Driven Development)**.** *"Spec-Driven Development is a software development approach in which specifications act as the primary artefact. These specifications are created before implementation, define what the system should do rather than how it should be implemented, and drive the implementation through AI-based programming agents that generate and verify implementations based on these specifications."*

Related software development approaches

Several existing software development approaches are mentioned in the literature in relation to Spec-Driven Development. While these approaches differ in purpose and level of abstraction, they share the idea that structured artifacts can guide software implementation.

One closely related concept is Formal Specification-Driven Development (FSDD), which was introduced in 2014 as a methodology emphasizing the use of formal design specifications as central development artifacts. In this approach, "*the design produces a formal specification in an organizationally selected behavioral specification language (JML, Dafny, Spec#, etc.), which becomes the artifact of record for both implementation and test agents*" [49]. Since FSDD was proposed prior to the widespread use of large language models, specifications were typically expressed in formal (behavioral) languages specifically designed for machine interpretation. These behavioral specification languages are designed to be interpreted by tools and compilers, enabling automated analysis and testing. Moreover, the approach assumes that developers manually implement systems based on these specifications, with automation primarily supporting verification rather than full code generation. In contrast, recent advances in AI-assisted software development have enabled systems to interpret natural-language specifications directly. As a result, Spec-Driven Development can rely on natural-language specifications that are both human-readable and machine-interpretable, allowing specifications to play a more direct role in guiding and generating implementations.

In addition to formal approaches such as FSDD, several development approaches focus on specific artifacts that guide implementation. The following approaches illustrate how different types of artifacts have been used to drive software development.

1. **Domain-Driven Design (DDD).** Domain-Driven Design is an approach to software development that emphasizes modeling software based on concepts from the problem domain. Domain-driven approaches aim to capture domain knowledge using abstractions and languages that reflect how domain experts describe the system. As described by [52], domain-driven software development seeks to use "languages that better capture the problem by using abstractions that are more familiar to experts in the domain". These domain-specific languages can then be transformed or refined into executable models or implementations through supporting tools. The approach relies on

principles such as abstraction, separation of concerns, and decomposition to structure complex software systems around domain concepts [47, 52].

2. **Model-Driven Development (MDD).** MDD often referred to as Model-Driven Engineering (MDE), is a software development paradigm that emphasizes the use of models as central development artifacts. MDE is commonly defined as an approach in which "abstract models of software systems are created and systematically transformed to concrete implementations" [22]. In this approach, models are treated as the main artifacts guiding development. A model itself represents "an abstraction of some aspect of a system" and is used to describe system behavior or structure at a higher level of abstraction [9, 22].
3. **Design by Contract.** Design by Contract is a software design approach in which the interaction between software components is governed by explicit behavioral contracts. These contracts define the responsibilities of each component and specify "what each party in the contract must guarantee for a correct call, and what each party is entitled to in return"[43]. Such contracts are typically expressed using assertions including preconditions and postconditions. "The precondition expresses requirements that any call must satisfy if it is to be correct", while a "postcondition expresses properties that are ensured in return by the execution of the call" [40, 43, 47, 49].
4. **Test-Driven Development (TDD).** Test-Driven Development is a software development approach in which automated tests are written before code is implemented. As described by [59], "before implementing production code, the developer writes automated unit test cases for the new functionality they are about to implement". Development then proceeds through short iterative cycles in which tests are written, executed to confirm failure, and satisfied through implementation. This test-first approach provides continuous feedback during development, enabling faults to be detected early as new code is added to the system [9, 47, 49, 55, 59].
5. **Behavior-Driven Development (BDD).** Behavior-Driven Development is an agile software development approach that emphasizes collaboration between technical and non-technical stakeholders during development. It is defined as "a method of software development that involves collaboration among both technical and non-technical team members working on a software product or project" [19]. BDD builds on ideas from Test-Driven Development but operates at a higher level of abstraction by shifting the focus from testing to the identification of the expected behaviors of a system. System behavior is typically specified using structured scenarios written in a "Given–When–Then" format that describe the context, event, and expected outcome of an interaction with the system [9, 19, 47, 49, 55].

Approach	**Driver of implementation**
Domain-Driven Design	Domain models and domain language
Model-Driven Development	System models
Design by Contract	Behavioral contracts
Test-Driven Development	Automated tests
Behavior-Driven Development	Behavioral scenarios
Spec-Driven Development	Specifications

Table 2.5: Software development approaches and the primary artifacts driving implementation.

Table 2.5 provides an overview of several software development approaches and the primary artifacts that drive software implementation in each approach. In contrast to these approaches, SDD positions specifications themselves as the driver of implementation. These specifications may incorporate multiple types of artifacts that are traditionally associated with different development approaches. As identified in Section 2.2.1, elements such as domain language, system models such as UML diagrams, contracts, and scenarios are described as components that may form part of a specification in SDD.

Workflow and Tools

To better understand how Spec-Driven Development operates in practice, the reviewed sources were also analyzed for descriptions of development workflows and supporting tools. The workflows identified in the grey literature are summarized in Table 2.6. Although the terminology and number of steps

differ across sources, two phases appear consistently across the described workflows: specification and implementation. These can be described as follows:

1. In the specification phase, the specification artifact describing what the system should do is created in collaboration with the programming agent. It acts as the primary artifact that guides the subsequent development process as the driver for implementation.
2. The implementation phase produces the system code based on the specification. In the workflows described in the literature, implementation typically takes the form of code generation using a programming agent, where the specification drives the generated implementation.

Most sources also mention a validation step [40, 45, 47, 55]. This validation is typically performed by a human developer who reviews generated plans, tasks, or specifications to keep a human in the loop. Validation could also be included in the workflow, where the programming agent validates its generated code by the specifications. In addition to these core phases, several sources include intermediate steps such as planning, design, or task decomposition [9, 17, 40, 47]. In these cases, the specification is translated into smaller tasks or development plans that structure the implementation process before code generation occurs.

Described workflow	**Constitution**	**Specify**	**Plan**	**Task**	**Implement**	**Validate**
specify→plan→implement→ validate [47]		X	X		X	X
Requirements→Design→Tasks→ implement [9, 17]		X	X	X	X	
Constitution→Specify→Plan→ Tasks→implement [9]	X	X	X	X	X	
Specify→Validate→Generate (Implement) [55]		X			X	X
Specify→Validate→Generate (Implement) [45]		X			X	X
Constitutional constraints across specification, planning, tasks, implementation, validate [40]	X	X	X	X	X	X

Table 2.6: Described workflow in grey literature

The literature also references tools that support SDD. Examples mentioned in the sources include GitHub Spec Kit and Amazon Kiro [9, 47, 55], Tessl [9, 47], Azure's Spec2Cloud [61], OpenSpec [12], and AIUP (AI Unified Process) [45], which provide structured AI workflows that translate specifications into implementation artifacts. An overview of these tools and their repository-level indicators is provided in Table 2.7. Among these, GitHub Spec Kit appears to be the most prominent implementation in practice, as suggested by indicators such as stars and forks. On GitHub, a star typically reflects user interest or endorsement of a repository, while a fork indicates that a user has created a copy of the repository, often to modify or extend it. Although these metrics do not directly measure real-world usage, they provide an indication of the relative visibility and community engagement of different tools. Research prototypes such as NL2POSTCOND [63] also explore the automatic derivation of postconditions or constraints from natural-language specifications.

In addition, several established testing and specification frameworks—such as BDD frameworks (Cucumber or SpecFlow) and TDD frameworks (RSpec, JUnit or pytest) are mentioned [47]. However, these frameworks do not fully align with SDD described in the reviewed sources, as they primarily focus on creating specifications for test cases rather than using a specification artifact to directly guide code generation.

Tool	Forks	Stars
GitHub Spec Kit [25]	6.7k	78.5k
Openspec [21]	4.4k	63.9k
Amazon Kiro [34]	175	3.2k
Spec2Cloud [5]	4	20
Tessl [54]	3	8
AIUP (AI Unified Process) [2]	2	11

Table 2.7: Overview of SDD-related tools and their repository-level indicators on GitHub.

2.2.1 The Spec in Spec-Driven Development

In addition to defining the concept, the literature is analyzed for descriptions of the specification artifact used in Spec-Driven Development. These descriptions are coded to identify common structural elements and content characteristics of such specifications. The 18 specification elements identified in Table 2.8 can be grouped into three categories: behavioral specification, formal contracts, and supporting artifacts. Appendix B shows a full mapping of these elements.

The largest group consists of behavioral specification elements. These describe what the system should do, including its functionality, expected behavior, and how it responds to different inputs. Examples include user stories, scenarios, input/output mappings, and edge cases. These elements show that SDD specifications focus strongly on describing system behavior in a structured and understandable way.

A second group consists of formal contracts. These include constraints, invariants, and preconditions and postconditions, which define conditions that must hold for the system to behave correctly. These elements appear less consistently across sources, but they introduce more precise ways of describing and checking system behavior. Their presence suggests that some SDD approaches incorporate ideas from formal methods and contract-based design.

The third group includes supporting artifacts, such as requirements documents, design artifacts, rationale, and domain language. These elements provide additional context and help explain or structure the specification. Some elements, such as implementation plans or task decomposition, are more related to the development process and do not seem to be included in all approaches.

Software Requirements Specifications

Earlier academic work on Formal Specification-Driven Development describes a workflow in which developers first review a Software Requirements Specification (SRS) before producing a more formal design specification using a formal specification language [49]. As discussed before, FSDD uses formal design specifications as they are machine interpretable. However, SDD uses natural language specifications as advances in AI-assisted software development (LLM code generation) have enabled systems to interpret natural-language specifications directly. Thus, there is a possibility that the natural language specifications that are used in SDD are standard software requirements specifications as used in FSDD before converting them to formal specifications. To better understand the structure of specifications used in Spec-Driven Development and whether they align with SRS, the specification elements identified in the literature (Table 2.8) were compared with the standard Software Requirements Specifications defined in IEEE 830 [1]. A detailed section-by-section mapping is provided in Appendix C.

The results from the mapping in Table 2.9 show substantial overlap between SDD specification elements and IEEE 830 SRS. Of the 18 elements, 7 have a direct counterpart and 8 a partial one, indicating that SDD specifications largely build on established SRS content rather than introducing an entirely new specification standard.

The direct matches cover core SRS elements such as behavior, functionality, constraints, and interface descriptions. The partial matches reflect cases where IEEE 830 addresses the same content but in a different format — for example, user stories and BDD scenarios express functional requirements that IEEE 830 captures through stimulus-response sequences and "shall" statements.

Three elements have no counterpart in IEEE 830. Invariants and preconditions/postconditions each appear in only a single source and may reflect the influence of Formal Specification-Driven Development rather than a core aspect of SDD itself. Implementation plan/task decomposition, mentioned by two

Specification element	Description	Source
Descriptive specifications		
behavior	Describing the intended behavior of the system: what it does, how it responds, focused on *what happens*, not *how*.	[9, 40, 47, 55, 63]
functionality	What the system should do: its functions, features, and capabilities.	[9, 40, 47, 55]
user stories with acceptance criteria	Requirements structured as user stories ("As a..." format) paired with acceptance criteria, sometimes including a Definition of Done.	[9, 47]
scenarios (BDD / Given-When-Then)	Behavioral scenarios expressed in Given/When/Then format, used to make expected behavior concrete and executable.	[47, 55]
business rules	Domain-specific rules that govern system behavior.	[47]
edge cases and error conditions	Specification of exceptional scenarios, boundary conditions, and how the system should handle errors.	[47]
input/output mappings	Explicit definitions of expected inputs and corresponding outputs, including structured examples.	[47, 55]
Formal specifications		
constraints	System, architectural, or business constraints	[40, 47, 55]
invariants	Properties that must always hold true throughout system operation.	[55]
preconditions and postconditions	Formal conditions that must hold before and after execution, capturing program behavior as verifiable assertions.	[63]
interface descriptions and integration contracts	Definitions of interface types, API contracts, and integration points between system components.	[47, 55]
state machines / sequential logic	Expected behavior through state machines or sequential logic.	[55]
formal specification	A formal specification in a behavioral specification language, such as JML, Dafny, or Spec#.	[49]
Supporting artifacts		
requirements document	Requirements captured as structured documents, such as detailed requirements documents or product requirements documents.	[9, 47]
rationale and intent	The *why* behind decisions, including declarations of intent and reasoning alongside constraints and patterns.	[40, 47, 63]
design artifacts	Supporting artifacts such as UML diagrams, narrative descriptions, and design documents.	[9, 49]
implementation plan / task decomposition	Plans detailing how to build features (technical requirements) and task lists that trace back to requirements.	[9, 40]
domain language	Use of a domain-oriented language to describe business intent.	[55]

Table 2.8: Specification elements identified in the literature.

sources, is explicitly excluded by IEEE 830, which separates the product specification from the development process (§4.8). However, its inclusion in SDD appears workflow-dependent rather than essential to the specification itself — some SDD workflows include planning or task decomposition as intermediate steps, while others may not. These three elements therefore represent possible extensions rather than definitive extensions from traditional SRS.

Overall, this alignment suggests that SDD specifications can be understood as natural-language SRS documents, rather than as an entirely new type of artifact.

2.3 Data-as-a-Product

Data is used within organizations in a variety of contexts, including reporting, business intelligence, and advanced analytical applications such as data science and machine learning. In the financial sector, for

Match	Specification elements	Count
Direct	behavior, functionality, edge cases and error conditions, input/output mappings, constraints, interface descriptions and integration contracts, requirements document	7
Partial	user stories with acceptance criteria, scenarios (BDD / Given-When-Then), business rules, state machines / sequential logic, formal specification, rationale and intent, design artifacts, domain language	8
None	invariants, preconditions and postconditions, implementation plan / task decomposition	3

Table 2.9: Results of specification element mapping (Appendix C) with IEEE 830 SRS as Direct, Partial or No match with IEEE 830 SRS standards.

example, data supports analytical processes related to transactions, customers, and operations. This data originates from multiple *(data) sources*, including transactional systems, user interactions, and digital channels. Data obtained directly from these sources is referred to as *raw data* which is data that has not been processed for use [18, 65]. Raw data can be distinguished based on its level of structure [18, 48]:

1. *Structured data* is organised according to a predefined schema in the form of a data table.
2. *Semi-structured data* does not follow a strict tabular schema but contains identifiable elements.
3. *Unstructured data* does not have a predefined format or does not fit in data tables, such as images, videos, recordings and emails.

Extract, Transform, Load

To enable the use of data from multiple sources and with varying structures, it is commonly processed through an *Extract, Transform, Load (ETL)* pipeline. In the *extract* step, data is retrieved from source systems. In the *transform* step, data is prepared through operations such as formatting, combining datasets, and aligning structures. In the *load* step, the resulting data is stored in a target system where it can be accessed for reporting, business intelligence, or advanced analytical use cases [18, 32, 33]. Once processed, data is stored in systems designed to support its use. Over time, different approaches to storing and managing data have been developed, reflecting differences in data types, processing approaches, and analytical requirements. These approaches are discussed in the following sections.

2.3.1 From Data Warehouses to Data Mesh

Building on the fundamental concepts of data sources, data types, and data processing, organizations have developed different architectural approaches to store, manage, and make data available for analytical use. These approaches have evolved over time in response to the increasing volume and variety of data and the growing demand for advanced analytical capabilities. Early approaches focused on centralized data warehouses, where structured data is integrated and prepared through ETL processes to support reporting and business intelligence. More recent developments introduced data lakes to accommodate a wider range of data types, followed by architectures that incorporate metadata and governance layers - data lakehouse. The most recent approaches, such as data mesh, emphasize domain-oriented data ownership and the concept of data as a product. Figure 2.2 illustrates this evolution from traditional data warehouse architectures to data mesh.

Data Warehouse

Data warehouses represent one of the earliest architectural approaches for storing and managing organizational data. They were introduced to integrate data from multiple data sources into a centralized repository, enabling a consistent and integrated view of data for analytical purposes [60]. The goal of data warehouses is to provide a "single version of truth," where data is structured, cleaned, and stored in a format optimized for reporting and business intelligence. Data warehouses primarily handle structured

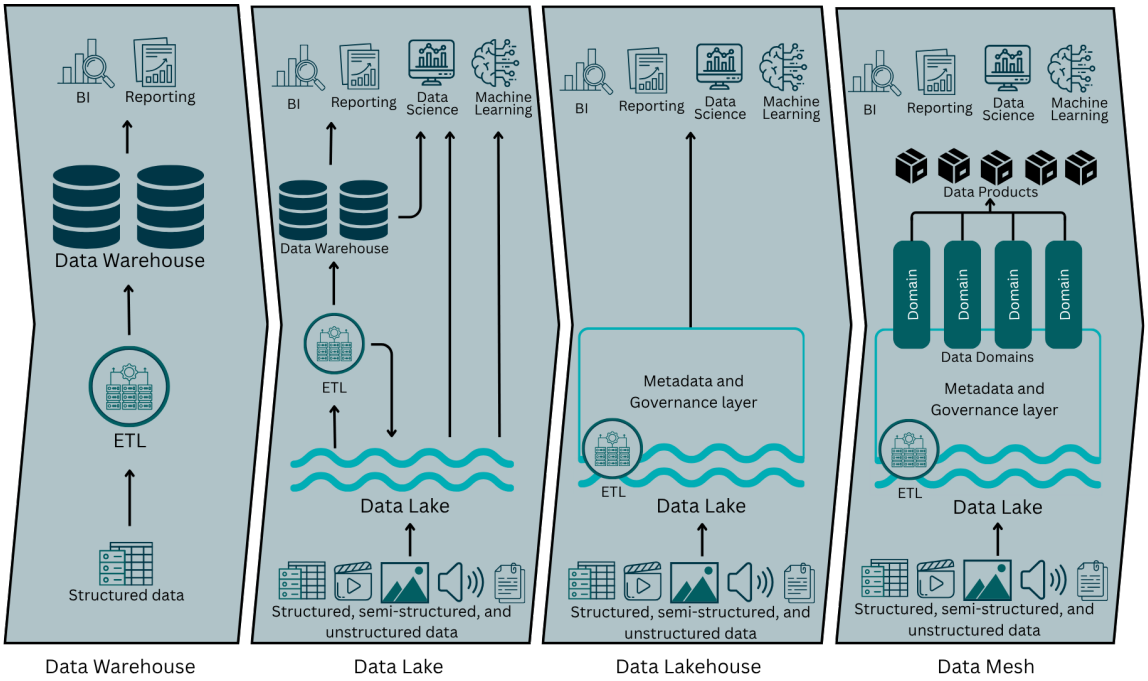


Figure 2.2: Evolution of data architectures: From Data Warehouse to Data Mesh [3, 7]

data and rely on ETL processes to transform and integrate raw data before it is loaded into the warehouse. This schema-on-write approach ensures that data is prepared in advance for specific analytical use cases [3, 51].

Data Lake

Data warehouses primarily handle structured data and, as volumes grew, faced limits in scalability and in handling unstructured data. To address this, data lakes were introduced as a more flexible storage architecture [3, 60]. Data lakes store structured, semi-structured, and unstructured data in raw form and apply a schema-on-read approach, where data is transformed only when needed [3, 60]. Within such architectures, data from the lake can be used directly for analytical purposes or be processed through ETL pipelines for further preparation. In practice, data lakes and data warehouses are often combined. Data can be extracted from the lake, transformed, and loaded into a data warehouse for structured reporting, or processed and stored back into the lake for reuse in other analytical workflows. This reflects a more flexible data flow where the data lake acts as a central storage layer supporting multiple processing paths. Data lakes therefore complement rather than replace data warehouses, which remain useful for structured data and standardized reporting [3, 60].

Data Lakehouse

The data lakehouse architecture extends the data lake by integrating capabilities traditionally associated with data warehouses into a single platform. This is introduced in a unified layer for metadata and governance—"the house" on top of the data lake [3, 51]. Compared to earlier two-tier setups (data lake plus data warehouse), the lakehouse can reduce the need for moving data between systems and for multiple ETL steps, allowing data to be accessed and processed within a more unified environment [51].

Data Mesh

In traditional centralized data architectures such as data warehouses and data lakes, data governance and ownership are centralized in a single team or department. As a result, data governance may become rigid and slow, in turn making it more difficult for the business to gain insights from its data [30, 38]. Data mesh aims to distribute governance and ownership across the organization by splitting data into different data domains, that usually correspond to the different business domains adding an extra layer to the data lakehouse architecture. Each data domain has its own Domain Data Owner and team, so that

Type	Description	Examples
Basic / fundamental	Dataset-level data products with domain-specific basic logic and broad reuse potential (domain models).	Datasets
Analytical / integrated	Data products that span multiple domains and enable cross-domain analysis.	Data APIs, reports
Advanced analytical	Data products with domain-specific analytical logic, typically designed for narrower use cases.	ML models, AI models, dashboards

Table 2.10: Types of data products

it is managed by experts who understand their own data in both a technical and business way [30, 60]. Different roles are described to be a part of data mesh: (i) Data Domain Owner is the responsible for data in a specific domain; (ii) Data Product Developer develops, maintains and serves data product domains but is not exclusive to one domain; (iii) Domain Architect is a subject matter expert for their data domain, the business needs of the domain, and its use cases; (iv) Federated Data Governance Manager manages the compliance of data products and domains with data governance policies and standards; and the (v) Data Consumer is the user of data products to support their business needs and use cases [26, 30, 38, 60].

2.3.2 What are data products?

Since Wang et al.'s seminal paper in 1998 [57] academics have discussed arguments for a product view on data and information, which data mesh incorporates [26]. In data mesh this view is referred to as data-as-a-product and executed as data products. Hasan et al. [26] review multiple existing definitions of data products and extend them based on case insights to provide a definition. This definition is used as the conceptual basis for understanding data products in this thesis, because it emphasizes their managed, consumable, and value-oriented nature.

Definition 2 (Data products [26])**.** *"A data product is a managed artifact that satisfies recurring information needs and creates value through transforming and packaging relevant data elements into consumable form."*

Additionally, Hasan et al. [26] derived five key characteristics of data products: (1) they satisfy recurring information needs, meaning they are created to address regularly occurring business needs of data consumers [36]; (2) they have a well-defined consumer base; (3) they create measurable value for organizations, either monetary or non-monetary [15]; (4) they are produced by combining data from multiple sources [11]; and (5) they are delivered in a consumable form, meaning they are packaged and made accessible in a way that meets specific user requirements [11, 41].

Data products can take different forms depending on their complexity (Table 2.10). The literature distinguishes between basic or fundamental data products, analytical or integrated data products, and advanced analytical data products. Basic or fundamental data products are dataset-level products that usually contain domain-specific logic and have the broadest reuse potential. Analytical or integrated data products combine or expose data across domains and support cross-domain analysis, for example through data APIs or reports. Advanced analytical data products contain more domain-specific analytical logic and are typically designed for narrower use cases, such as machine learning models, artificial intelligence models, dashboards, or other analytical applications [26, 30].

Data product lifecycle, data catalog, and data shop

Karkošková [30] proposes a data product lifecycle that describes the stages a data product goes through from initiation to refinement or decommissioning. As shown in Figure 2.3, the lifecycle starts with data product initiation, where the use case, consumer, product owner, and data domain are identified. The data product is then built, published, requested, delivered, consumed, and either refined or decommissioned.

The lifecycle also highlights the role of the data catalog and data shop. The data catalog acts as a central repository of metadata and information about available data products. It supports discovery and understanding by describing data products through elements such as business and technical metadata,

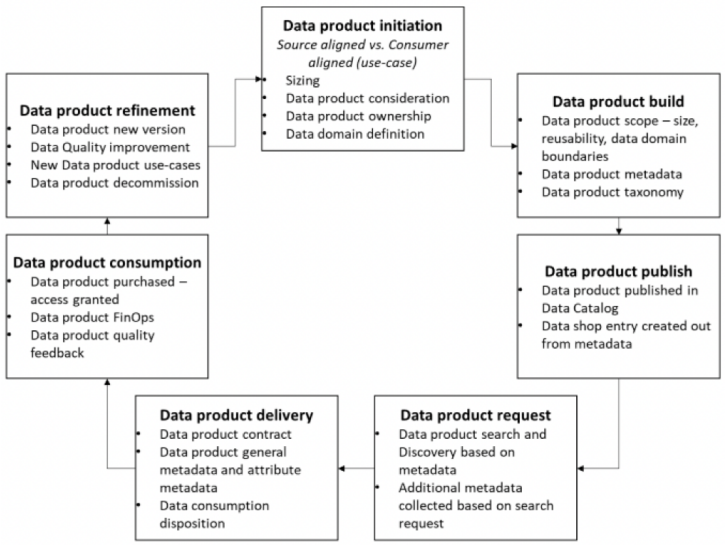


Figure 2.3: Data product lifecycle, reproduced from [30]

data quality information, data dictionary entries, business glossary terms, and data lineage. The data shop can be understood as the consumer-facing part of this environment, where users can search for data products, review associated data contracts and usage terms, and request access to the data product. In this way, the data catalog supports the management and description of data products, while the data shop supports their discovery, request, and consumption [26, 30].

Technical components of data products

In addition to conceptualising data products in terms of their definition, types, lifecycle, and role in the data catalog, a literature review is conducted to identify the technical components of data products. To identify the technical components of data products, six sources are reviewed and coded [8, 26, 30, 38, 53, 60]. Among the reviewed sources, three provide explicit technical or component-based definitions of data products see Table 2.11. Table 2.12 summarizes the identified components, their descriptions, and the sources in which they appear. Appendix D shows the complete technical components and quotes.

Technical definition	Source
"A data product can be considered as combination of data, transformation logic, data quality, metadata, SLA management, identity and access mechanism and change and incident management that is made available to other users as a consumable service or product."	[30]
"A data product is a unit of logical architecture which comprises and controls several structural components such as data, code, policies and infrastructure dependencies with the overall goal to share data as a product frictionless and autonomously."	[60]
"They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage"	[8]

Table 2.11: Technical definitions of data products

Component	Description	Source
Data	The data component refers to the data contained within the data product. The literature describes data as a central structural component of a data product and notes that it may exist in different forms, such as batch files and events.	[8, 26, 30, 38, 53, 60]

Metadata	Metadata refers to the information associated with the data in a data product. It supports the description, interpretation, and semantic documentation of the data, and is identified as one of the components that facilitates data sharing and usage.	[8, 30, 38, 53]
Code	Code refers to the software elements that support the data product. This includes code for data pipelines, applications that enable access to data and metadata, and code related to access policies.	[8, 26, 38, 60]
Transformation logic	Transformation logic refers to the rules or processing steps through which source data is transformed into the data product. The literature indicates that data may be sourced from one or more domains and processed through different transformations.	[30, 53]
Infrastructure	Infrastructure refers to the technical dependencies and environment that enable the data product to operate. It supports access to data and metadata, the execution of code, and the technical functioning of data pipelines.	[8, 38, 53, 60]
Policies	Policies refer to the rules established for a data product. These policies can be defined at a detailed level and are identified as one of the structural components contained within a data product.	[8, 60]
Data quality	Data quality refers to the quality-related checks and expectations associated with the data product. The literature links data products to data quality checks and describes data quality as part of trustworthiness.	[26, 30, 53]
SLA/SLO management	SLA/SLO management refers to the explicit service-level expectations associated with a data product. The literature refers to SLA management and the implementation of explicit SLOs for data products.	[30, 53]
Identity & Access Mechanism	The identity and access mechanism refers to the way access to the data product is controlled and assigned. The literature explicitly refers to identity and access mechanisms and access control.	[26, 30]
Incident Management	Incident management refers to the mechanisms for handling incidents related to the data product. The literature identifies incident management as one of the components of a data product.	[30]

Table 2.12: Technical components of data products

The literature also describes several characteristics that data products should exhibit, including being discoverable, addressable, understandable, accessible, interoperable, reusable, secure, and valuable [26, 30, 38, 60]. These characteristics partly overlap with the FAIR principles, which provide guidance for making digital resources Findable, Accessible, Interoperable, and Reusable for machines and humans [28]. Discoverability closely relates to Findability, while accessibility, interoperability, and reusability directly correspond to three of the four FAIR principles. The remaining qualities extend beyond the four categories defined by FAIR. These characteristics are not treated as technical components in Table 2.12, because they describe properties of a data product rather than elements of which it consists. However, they help explain the role of the technical components: metadata supports discoverability and understandability, while identity and access mechanisms support addressability, accessibility, and security.

The conceptual definition by Hasan et al. explains the purpose of data products, while the coded literature review clarifies how data products are technically operationalised. Based on this synthesis, the following technical working definition is used:

Definition 3 (Technical working definition of data products)**.** *A data product is a managed artifact that combines data, metadata, code, transformation logic, infrastructure, policies, data quality mechanisms, SLA/SLO management, identity and access mechanisms, and incident management to make data available as a consumable product. A data product should be discoverable, addressable, understandable, accessible, interoperable, reusable, secure, and valuable, and is managed through a lifecycle in which it is built, published, requested, consumed, and refined or decommissioned.*

2.4 Case context

Van Lanschot Kempen is a Dutch financial services organisation in the wealth management sector, specialising in private banking, investment management, and investment banking [56]. This research is situated within its Data Management department, which is responsible for data governance and management across the organisation. As part of its data management approach, Van Lanschot Kempen applies data mesh principles, in which data is organised across business domains and data products make domain data available for consumption. Each data product is owned by a specific domain, although it may combine or depend on data from multiple domains. This study focuses specifically on analytical or integrated data products, which combine or expose data across domains to support analytical and reporting purposes.

2.4.1 Data product lifecycle

At Van Lanschot Kempen, data products follow the lifecycle introduced in Section 2.3.2, which extends from initiation to refinement or decommissioning. The build phase takes place in Databricks, where data products are developed and deployed. After development, data products are published to the data catalog, where they become visible to potential consumers. Consumers can subsequently search for a data product and request access. The relevant domain owner reviews the request based on its intended use and applicable governance considerations. After approval, the requester can consume the data product. Feedback, changing requirements, or new use cases may then lead to its refinement.

2.4.2 Data catalog

The data catalog is the central application through which data products are published, discovered, and requested. It contains information about the available data products and provides lineage information that allows users to understand where data originates and how it flows through the organisation. The data catalog is connected to the organisation's Corporate Data Model (CDM), which structures the data domains and provides descriptions of business concepts. Van Lanschot Kempen has approximately 40 data domains, grouped into the super domains Client Relations, Investments, Loans, Other, and Van Lanschot Kempen. Examples of individual domains include Loans, Investments, Affluent Clients, and Asset and Liability Management.

2.4.3 Data product development

The technical development of data products takes place in Databricks. Source data that can serve as input for a data product is made available through Unity Catalog, which organises and governs access to data within Databricks.

Data products are implemented using Declarative Automation Bundles (DABs). A DAB groups the source files and configuration used to define and deploy Databricks resources, such as jobs and pipelines. A DAB template provides a reusable starting structure from which a bundle project can be created. Developers can use the Databricks command-line interface (CLI) to validate, deploy, and run the bundle [13]. Within the case organisation, developers begin with a predefined DAB template. This template provides the initial project structure and configuration within which the code, transformation logic, Databricks resources, and deployment settings of a data product are developed. Developers edit these projects in Visual Studio Code and store them in an Azure repository. GitHub Copilot is available in this environment as a programming agent supporting AI-assisted development.

Table 2.13 maps the technical data-product components identified earlier in this chapter to their implementation in the case organisation. It shows that the DAB contains the code, transformation logic, and infrastructure configuration, while other components are supported by Unity Catalog, the Corporate Data Model, and organisation-wide governance and operational processes. Figure F.1 presents the folder structure of this template. The root configuration files define the bundle and Python package, while the `resources` directory contains the Databricks resource definitions. The `src` directory contains the implementation, transformations, notebooks, and tests. The same template provides the initial project structure used during the tool comparison, demonstration, and controlled evaluation.

Development also follows organisation-specific technical and governance requirements. The development time of a data product depends on its size and complexity: smaller data products may take approximately

three weeks to develop, while larger or more complex products can take up to three months.

Component	**Case-specific implementation**
Data	Source data available through Unity Catalog serves as input for data products. Each data product is owned by one domain but may use or combine data from multiple domains.
Metadata	The CDM provides the business definitions used within the organisation. Data elements in Unity Catalog are linked to the relevant CDM concepts through tags.
Code	Code is edited in Visual Studio Code, stored in an Azure repository, and included in the Declarative Automation Bundle used to build and deploy the data product. GitHub Copilot is available to support AI-assisted coding.
Transformation logic	Transformation logic is implemented in code and executed in Databricks to transform source data from Unity Catalog into the resulting data product.
Infrastructure	Databricks provides the main environment for building and deploying data products. Declarative Automation Bundles package the required resources and deployment settings, while the associated files are stored in an Azure repository.
Policies	Organisation-wide data governance policies and development requirements apply to the development and use of data products. Unity Catalog supports their technical enforcement through permissions and access controls.
Data quality	Data-quality requirements are addressed through checks implemented in the Databricks pipelines used to build data products.
SLA/SLO management	SLA/SLOs are defined for specific use cases rather than as a standard component of every data product. Where applicable, delivery expectations are reflected in the data-product implementation.
Identity & access mechanism	Access is requested through the data catalog and reviewed by the relevant domain owner. Technical permissions are enforced through roles and access controls in Databricks.
Incident management	Data-product incidents are handled through the organisation-wide data incident management standard rather than through a separate process for each data product.

Table 2.13: Mapping of technical data-product components to the case organisation

3 Method

This chapter describes the research approach used to design and evaluate the proposed Spec-Driven Development tool. It first explains the application of Design Science Research and the activities performed within each stage. It then describes the research context and scope, followed by the methods used for data collection and analysis.

3.1 Design Science Research

This research follows a Design Science Research (DSR) approach. DSR is a research methodology within Information Systems that addresses practical problems through the design and evaluation of artefacts while also contributing to scientific knowledge. An artefact may take the form of a model, method, framework, or software solution and is evaluated in relation to its intended purpose and usefulness [27, 58].

DSR is suitable for this thesis because the research combines the emerging concept of Spec-Driven Development with the specific requirements of data-product development in an organisational setting. The study aims to create an artefact that supports a more structured application of AI-assisted development and to evaluate this artefact with practitioners in the case organisation. The iterative character of DSR supports the development and refinement of the artefact based on insights obtained during its demonstration and evaluation.

The artefact in this thesis takes the form of a Spec-Driven Development tool for AI-assisted data-product development. The tool operationalises SDD by supporting a structured development workflow. This workflow forms part of the tool's design and describes the development process supported by the artefact. Its specific structure and components emerge from the design and development activities.

To structure the research process, this thesis applies the DSR methodology proposed by Peffers et al. [46]. This methodology consists of six stages: (1) problem identification and motivation, (2) objectives of a solution, (3) design and development, (4) demonstration, (5) evaluation, and (6) communication.

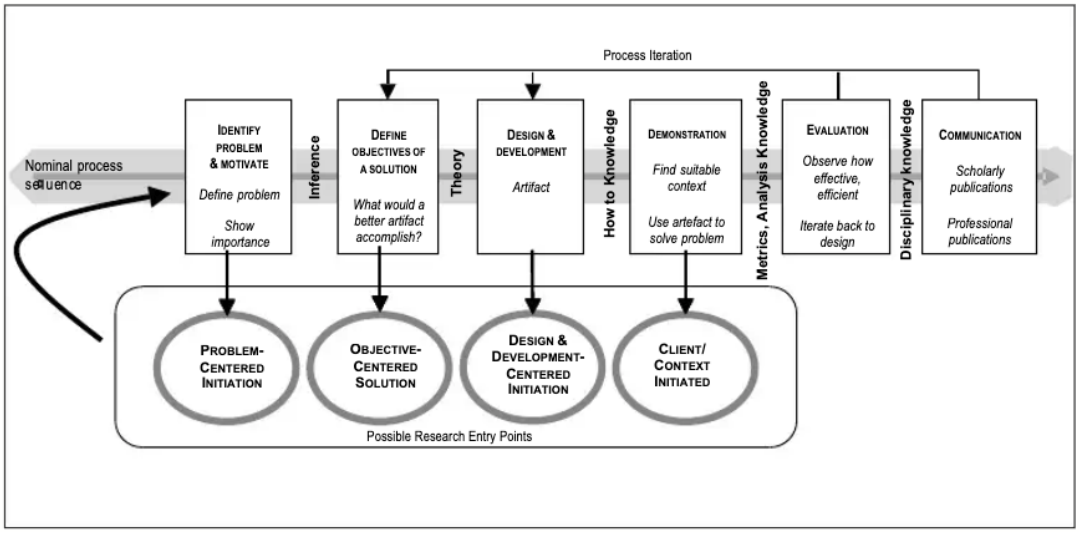


Figure 3.1: Design science research process (DSRP) model [46]

Figure 3.1 visualises these stages and shows that the DSR process is iterative rather than strictly linear. Insights obtained during later stages can lead to the reconsideration of earlier design decisions. In this thesis, iteration occurs primarily between the *Design and development*, *Demonstration*, and *Evaluation* stages. An initial version of the tool is developed and applied to a synthetic case during the demonstration. Findings from this application inform refinements before the tool is evaluated with practitioners.

The results of the controlled evaluation subsequently inform a refined version that is applied to ongoing development tasks during the practical evaluation.

The six DSR stages are applied in this thesis as follows.

3.1.1 Problem identification and motivation

The first stage identifies and motivates the research problem addressed by the thesis. As described in Section 1.1, SDD is an emerging approach that primarily appears in practitioner and grey literature. A consistent formalisation remains lacking, and limited academic research examines how SDD principles can be translated into concrete tools for specific development contexts.

This limitation is particularly relevant to data-product development, where a programming agent requires information about data sources, transformations, metadata, infrastructure, data quality, security, and access requirements. General-purpose SDD tools do not necessarily account for these domain-, platform-, and organisation-specific requirements. The research problem therefore concerns the absence of an established and evaluated approach for applying SDD to the technical specification and AI-assisted development of data products.

3.1.2 Objectives of a solution

The second stage defines the objective of the proposed solution based on the identified research problem. As presented in Section 1.2, the objective of this thesis is to design and evaluate a Spec-Driven Development tool that supports AI-assisted data-product development. The tool focuses on analytical or integrated data products and on the initiation and build phases of the data-product lifecycle. The design aims to support more structured AI-assisted data-product development.

The objective is further translated into design requirements during the design and development stage. These requirements are derived from the conceptualisation of SDD, the characteristics of SDD specifications, the technical components of data products, the analysis of existing SDD tools, and the development context of the case organisation.

3.1.3 Design and development

During the third stage, presented in Chapter 4, the Spec-Driven Development tool is designed and developed. The design process combines findings from the literature and tool comparison and translates them into design requirements and a working implementation. The process consists of six main activities.

1. Identification of design inputs. The conceptual inputs originate from the literature reviewed in Chapter 2. Practitioner and grey literature on SDD is coded to identify recurring definition elements, workflow phases, and characteristics of SDD specifications. Academic literature on data products is reviewed and coded to identify their technical components, while the IEEE 830 Software Requirements Specification structure provides an established reference for the specification content. Case-specific inputs are obtained from organisational documentation and the existing data-product development environment within the case organisation.

2. SDD tool comparison. The tools identified through the literature review are analysed in two stages. First, the public documentation of Spec Kit, AIUP, Kiro, Tessl, Spec2Cloud, and OpenSpec is reviewed to describe how each tool provides its workflow and assess its suitability for inclusion in a comparison using GitHub Copilot. A tool is considered suitable when it can be used directly or adapted without changing its core workflow.

Second, Spec Kit, AIUP, Tessl, Spec2Cloud, and OpenSpec are applied to the same synthetic data-product case. Each tool is initialised in a separate copy of the same Declarative Automation Bundle (DAB) template. The template provides every tool with the same initial folder structure, Databricks configuration, and project files. The structure of this template is presented in Figure F.1. The same case description and source data are also used for every application. All applications use GitHub Copilot with Claude Sonnet 4.6 as the language model and the medium reasoning setting. This keeps the initial project context, programming-agent environment, and underlying model constant across the comparison. Each tool is configured and used according to its public instructions. The synthetic case uses the Databricks New York taxi sample data and is presented in Appendix E.1.

3. Tool selection. The comparison is used to select an existing tool as the basis for the artefact. OpenSpec is selected because its lightweight and modular structure makes it suitable for extension. Its suitability is explored by adding a Databricks-specific skill and applying the adapted version to the same synthetic case. The baseline and adapted versions are compared using the relevant criteria from the tool comparison. This activity assesses whether platform-specific knowledge can improve contextual grounding.

4. Analysis of Software Requirements Specifications. After OpenSpec is selected as the basis for the artefact, its proposal and specification artefacts are mapped to the sections of the IEEE 830 Software Requirements Specification structure. The technical components of data products identified in the literature are subsequently mapped to the SRS sections needed to specify those components. The two mappings are compared to identify which requirements categories relevant to data-product development are already covered by OpenSpec and which are missing or require additional support.

5. Derivation of design requirements and tool architecture. The gaps identified through the SRS analysis are combined with the design insights from the tool comparison and the Databricks-skill experiment. Together, these findings are consolidated into design requirements for the artefact. Each design requirement is then translated into one or more components of the tool architecture.

6. Artefact development. The resulting architecture is implemented as an extension to OpenSpec in GitHub Copilot. The components are implemented as skills using the native `/skill-creator` functionality. These include skills for clarification, extending the proposal and specifications, validation, configuration, Databricks-specific guidance, and organisation-specific development requirements. The OpenSpec prompts and skills are adapted where necessary so that these components are invoked at the intended points in the workflow. The implementation is tested iteratively while the components are created and integrated.

3.1.4 Demonstration

The fourth DSR stage demonstrates the implemented artefact through an end-to-end application by the researcher. The demonstration starts in a new copy of the same DAB template used during the tool comparison and uses the same synthetic case (Appendix E.1). This allows the proposed extension to be assessed in the same development context as the baseline tools. The researcher executes the workflow from the initial request to implementation and records observations about its operation.

Observations from the first demonstration are used to refine the artefact, after which the demonstration is repeated using the same synthetic case. The refined artefact is also compared with baseline OpenSpec and OpenSpec with the Databricks skill using the relevant criteria from the earlier tool comparison. This stage verifies that the artefact operates as intended and that its components are correctly integrated before it is evaluated by practitioners.

3.1.5 Evaluation

The fifth stage evaluates the artefact in two parts: a controlled evaluation and a practical evaluation.

The controlled evaluation consists of four individual one hour sessions with practitioners from the case organisation. Each participant works in a separate copy of the same DAB template. All participants use the same controlled organisational case, supplemented with source information derived from organisational data domains. The complete case description is presented in Appendix E.2. Each session uses a concurrent think-aloud protocol, in which participants verbalise their thoughts while using the tool to complete the assigned task [29]. Participants are specifically asked to express their expectations and uncertainties and to comment on generated outputs as they occur. The researcher observes the interaction, records participants' comments, and only prompts them when they stop verbalising.

Following the controlled task, each participant completes an adapted Technology Acceptance Model (TAM) questionnaire [14]. TAM assesses two constructs related to technology acceptance: perceived usefulness and perceived ease of use. The questionnaire is based on the six-item scales developed by Davis for each construct, with the wording adapted to refer to the proposed tool and data-product development. One additional statement captures participants' willingness to use the tool in practice. TAM has also been applied in the formative evaluation of an early-stage technology prototype with a small participant group to obtain initial feedback on technology acceptance [50]. All thirteen statements

use a five-point Likert scale ranging from 1 (*strongly disagree*) to 5 (*strongly agree*). The complete questionnaire is presented in Table G.1.

The observations from the controlled evaluation are used to refine the tool before the practical evaluation. In the practical evaluation, two participants from the controlled evaluation apply the refined tool within the existing DAB projects associated with their own ongoing development tasks during a joint two hour session. The researcher remains present to observe how the workflow is used and to record participant comments and feedback. The practical evaluation examines the applicability of the refined tool to ongoing organisational development tasks. Following the practical evaluation, P2 and P3 complete the same questionnaire again. Their responses are compared descriptively with their responses following the controlled evaluation.

3.1.6 Communication

The research process, artefact, and findings are documented and communicated through the final thesis.

3.2 Research context and scope

The research is conducted at Van Lanschot Kempen and is situated within the Data Management department. The study focuses on the development of analytical or integrated data products in Databricks. These products combine or expose data across domains for analytical and reporting purposes.

The controlled evaluation involves four practitioners whose roles include data-product development within the case organisation. Participants are selected based on their involvement in this work and their relevance to the intended users of the artefact. The group includes variation in Databricks and GitHub Copilot experience and in typical development approach, as shown in Table 3.1. Two of these participants subsequently take part in the practical evaluation, in which they apply the refined artefact to their own ongoing development tasks.

Participant	**Role**	**Databricks experience**	**GitHub Copilot experience**	**Typical development approach**
P1	Senior data engineer	Experienced	Limited	Development-first
P2	Senior data engineer	Experienced	Limited	Development-first
P3	Senior data engineer	Experienced	Beginner	Mixed
P4	Business analyst & data engineer	Beginner	Beginner	Specification-first

Table 3.1: Overview of evaluation participants

The study focuses on the technical specification and initial development of data products during the initiation and build phases of the data-product lifecycle. The evaluation covers specification and the initial transition towards implementation. Completion and deployment of full data products and the subsequent lifecycle stages of publication, delivery, consumption, refinement, and decommissioning fall outside the evaluated scope.

The study uses existing LLM-based development technology through GitHub Copilot and does not train or develop a new AI model. The technical implementation and evaluation are specific to the examined GitHub Copilot, OpenSpec, and Databricks environment.

3.3 Data collection

The study primarily collects qualitative data, complemented by quantitative responses from the TAM questionnaire. Data are collected throughout the design, demonstration, controlled evaluation, and practical evaluation.

3.3.1 Design and demonstration data

The design stage uses the academic, practitioner, and grey-literature sources reviewed in Chapter 2, the IEEE 830 Software Requirements Specification standard, and the public documentation and repository contents of the identified SDD tools.

During the tool comparison, the researcher records the setup process, interactions with the programming agent, generated artefacts, introduced assumptions, and implementation behaviour. The generated artefacts and recorded observations provide the material used to compare the tools.

During the demonstration, the collected material consists of the generated proposal, specification, design, task, and implementation artefacts, together with researcher observations about the execution of the workflow. These observations record whether the intended components are invoked, whether the expected artefacts are produced, and where technical issues or manual intervention occur. The same types of data are collected when the demonstration is repeated after refinement.

3.3.2 Controlled-evaluation data

During the controlled evaluation, qualitative data are collected from the comments and feedback verbalised by participants during the individual think-aloud sessions and from researcher observations documented in session notes. The researcher notes form the primary and consistent record across the four sessions. Audio recordings, where available, serve as a supplementary record to support the completeness of these notes. Following each session, participant responses to the thirteen TAM statements are collected.

3.3.3 Practical-evaluation data

During the practical evaluation, data are collected while two participants apply the refined tool to their own ongoing development tasks. The collected data consist of participant comments and feedback, researcher observations, and the workflow stages completed during each session. The practical-evaluation applications are documented through researcher notes. Following the session, participant responses to the thirteen TAM statements are collected.

3.3.4 Data handling

Before each evaluation session, participants are informed about the purpose and procedure of the evaluation and provide consent to participate. Evaluation data are reported using the participant identifiers P1–P4. Organisationally sensitive details contained in the development tasks and generated outputs are not reported in the thesis; where relevant, they are described at a higher level of abstraction. The collected materials are stored in accordance with the requirements of the case organisation.

3.4 Data analysis

The literature is analysed through qualitative coding, the tool comparison and demonstration through structured comparison, and the evaluation data through the grouping of recurring observations. The TAM questionnaire responses are analysed using descriptive statistics.

3.4.1 Analysis of design data

The practitioner and grey-literature sources concerning SDD are coded to identify recurring definition elements, workflow phases, and characteristics of SDD specifications. The academic literature on data products is coded to identify recurring technical data-product components. The identified SDD specification elements are compared with the IEEE 830 Software Requirements Specification structure, while the technical data-product components are mapped to the SRS sections required to specify them.

The documentation analysis compares the identified SDD tools against the workflow phases derived from the literature. Support for each phase is classified as explicit, partial, or not present based on the documentation and repository contents of the respective tools.

For the tool comparison, the recorded observations and generated artefacts are compared across the five tools. Recurring differences are grouped into the eight comparison criteria presented in Chapter 4. Each

tool is assigned a qualitative assessment of high, medium, or low for each criterion based on the observed tool behaviour and generated outputs. These assessments support comparison between the tools. The findings from the tool comparison are translated into design insights for the proposed artefact.

The OpenSpec artefacts are subsequently mapped to the IEEE 830 SRS sections. This mapping is compared with the SRS sections identified as relevant to the technical components of data products. The resulting coverage gaps are combined with the findings from the tool comparison and the Databricks-skill experiment to derive the design requirements for the artefact.

3.4.2 Analysis of demonstration data

The observations from the initial demonstration are reviewed to identify technical and workflow issues that affect whether the artefact operates as intended. Each identified issue is linked to the relevant tool component and translated into a refinement. The repeated demonstration is used to determine whether the identified issues have been addressed. The refined artefact is also compared with baseline OpenSpec and OpenSpec with the Databricks skill using the relevant criteria from the earlier tool comparison. This comparison assesses how the successive versions differ in their support for the intended workflow.

3.4.3 Analysis of controlled-evaluation data

Researcher observations and participant comments from the think-aloud sessions are reviewed and grouped according to recurring themes. These themes are derived from the collected material rather than fixed in advance. Similarities and differences between participants are examined, and participant identifiers are used to indicate which sessions provide evidence for each finding. Observations that point to limitations in the tool are linked to the relevant components and used to determine refinements.

For the TAM questionnaire, the arithmetic mean is calculated for each of the thirteen statements across the four participants. The item-level averages are reported for perceived usefulness, perceived ease of use, and willingness to use. Because of the small number of participants, the questionnaire results are used descriptively to complement the qualitative findings.

3.4.4 Analysis of practical-evaluation data

The two practical applications conducted during the joint session are analysed separately. Researcher observations and participant feedback are reviewed to determine how the workflow is followed, how participants assess the relevance of the generated outputs to their task contexts, and whether their experiences indicate a need for further refinement. The observations and feedback from the two applications are then compared to identify recurring or contrasting experiences. This analysis provides evidence about the applicability of the artefact to the examined organisational tasks.

4 Design

This chapter presents the design and development of the proposed Spec-Driven Development tool for supporting the initial development of data products. The design combines the conceptual findings from the literature review, observations from existing SDD tools, the IEEE 830 Software Requirements Specification structure, the technical components of data products, and the development context of the case organisation.

The chapter first analyses the documentation of the identified SDD tools to determine whether they are suitable for inclusion in a shared comparison using GitHub Copilot. The suitable tools are then applied to the same synthetic data-product development case to compare their behaviour and generated artefacts. This comparison supports the selection of OpenSpec as the foundation for the proposed artefact and provides design insights for its extension. The suitability of OpenSpec for extension is subsequently examined by adding a Databricks-specific skill. The OpenSpec artefacts are then mapped to the IEEE 830 SRS sections and compared with the sections required to specify the identified data-product components. The insights from the tool comparison and the gaps identified through the SRS analysis are consolidated into design requirements and translated into the architecture and implementation of the proposed tool. The technical demonstration and evaluation of the implemented artefact are presented in the following chapter.

4.1 Analysis of existing SDD tools

This section compares the SDD tools identified in Chapter 2. The comparison consists of two stages. First, the public documentation of Spec Kit, AIUP, Kiro, Tessl, Spec2Cloud, and OpenSpec is analysed to describe their workflows and to determine whether they are suitable for inclusion in a shared comparison using GitHub Copilot. Second, the suitable tools are applied to the same synthetic data-product development case. Their behaviour and generated artefacts are compared to select a foundation for the proposed artefact and derive insights for its design.

4.1.1 SDD tools

Table 4.1 compares the six tools against the SDD workflow phases identified in the literature: constitution, specification, planning, task decomposition, implementation, and validation. A phase is marked as explicitly supported when the documentation describes a corresponding skill or workflow step. Partial support indicates that the tool provides related functionality but does not represent it as a separate workflow phase.

Tool	**Constitution**	**Specify**	**Plan**	**Task**	**Implement**	**Validate**
Spec Kit	X	X	X	X	X	X
AIUP		X			X	
Kiro	∼	X	X	X	X	∼
Tessl		X			X	
Spec2Cloud		X	X	∼	X	X
OpenSpec		X		X	X	

Table 4.1: Documented workflow coverage of SDD tools (X = explicit support, ∼ = partial or implicit support, empty = not present)

The comparison shows that specification and implementation are the only phases explicitly supported by all six tools. This corresponds with the two core phases of SDD identified in Chapter 2. The main differences concern whether the tools also support constitution, planning, task decomposition, and validation.

Spec Kit is the only tool that explicitly supports all six phases. Kiro explicitly supports planning and task decomposition, while its support for constitution and validation is partial. Spec2Cloud explicitly supports planning and validation and provides partial support for task decomposition. AIUP and Tessl explicitly support only specification and implementation. OpenSpec explicitly supports specification, task decomposition, and implementation, while planning and validation are not represented as separate phases. Beyond specification and implementation, the documented workflow structure therefore differs considerably between the tools.

Workflow coverage alone does not show whether a tool can be used within the shared GitHub Copilot environment. The documentation is therefore also analysed to determine how each workflow is provided and whether it can be used while keeping the programming agent and language model constant. The results are presented in Table 4.2.

Tool	**How the workflow is provided**	**Use with GitHub Copilot**	**Selected**
Spec Kit	User-invoked skills for use with a programming agent	Directly usable	Yes
AIUP	Claude plugin containing user-invoked skills	Usable after adaptation	Yes
Kiro	Orchestrated workflow in a dedicated IDE with a built-in programming agent	Not usable through GitHub Copilot	No
Tessl	Orchestrated skills for use with a programming agent	Directly usable	Yes
Spec2Cloud	Orchestrated skills for use with a programming agent	Directly usable	Yes
OpenSpec	User-invoked skills for use with a programming agent	Directly usable	Yes

Table 4.2: Documented form of the SDD tools and selection for further comparison

Table 4.2 distinguishes between user-invoked and orchestrated skills. In a user-invoked workflow, the user starts each activity separately through a slash-prefixed prompt in the programming-agent chat. For example, the OpenSpec proposal activity is started through `/opsx-propose`, after which the programming agent loads and follows the corresponding skill. In an orchestrated workflow, the user starts a broader workflow that coordinates which skills are used and in which order. The user therefore does not need to start each skill through a separate prompt.

The table also shows which tools can be used within a GitHub Copilot environment. Spec Kit, Tessl, Spec2Cloud, and OpenSpec can be used directly with GitHub Copilot. AIUP is provided as a Claude plugin, which is an installable extension containing skills designed for Claude Code. These skills can be adapted for use with GitHub Copilot without changing the main AIUP workflow. Kiro instead provides its workflow through a dedicated IDE with a built-in programming agent. Using Kiro through GitHub Copilot would not be possible as GitHub Copilot is not part of Kiro's IDE. Kiro is therefore not included for further comparison, while the other five tools are. This selection allows the same programming agent and language model to be used for each tool.

The following subsection applies the five selected tools to the same synthetic data-product development case. Their behaviour and generated artefacts are compared to select a suitable foundation for the proposed artefact and derive insights for its design.

4.1.2 Tool comparison

Following the documentation analysis, each selected tool is initialised in a separate copy of the same DAB template and applied to the same synthetic data-product case. The case concerns the development of a reusable Taxi Trip Reporting data product based on NYC taxi trip data for reporting in Power BI. It supports analysis of trips over time, distance, duration, and fare amounts, and requires decisions about relevant source data, attributes, transformations, and the resulting output structure. The complete

case description is presented in Appendix E.1. The comparison setup is described in Section 3. The same case description is provided as the initial input for each tool. Each tool's documented workflow is then followed until it produces its artefacts and, where supported, begins implementation. During each application, observations are recorded about the setup process, interactions with the programming agent, generated artefacts, introduced assumptions, and implementation behaviour. To structure the analysis, these observations are organised according to the comparison criteria presented in Table 4.3. The table uses qualitative ratings to summarise the relative differences between the five tools. The qualitative ratings show how each tool performs relative to the other tools for each criterion. A higher rating indicates more favorable performance, while a lower rating indicates less favorable performance. The remainder of this subsection explains the observations on which the comparison is based.

Tool	1. Setup ease	2. Databricks compatibility	3. Assumption accuracy	4. Clarification support	5. Data model clarity	6. Progress tracking	7. Adaptability	8. Validation support
Spec-kit	Low	High	Medium	Low	High	Medium	Medium	High
OpenSpec	High	Medium	Low	Low	Medium	Medium	High	Low
AIUP	Medium	Low	Medium	Low	High	Low	Low	Low
Tessl	High	Medium	Low	Medium	Medium	Medium	Medium	Low
Spec2Cloud	Low	Medium	Medium	High	High	High	Medium	High

Table 4.3: Comparison of Spec-Driven Development tools based on observed workflow characteristics

1. Setup ease

Setup ease describes the effort required to configure a tool and complete its documented workflow before implementation begins. During the comparison, the researcher assesses this criterion based on the number of workflow steps that must be initiated, the manual preparation required, the number of generated files that require review, and the elapsed time between providing the initial case description and reaching implementation. The reported times include interaction with the programming agent, review of generated artefacts, and waiting for responses. Appendix H provides a complete overview of the files generated or modified by each tool. The counts discussed below include files produced before implementation and exclude implementation files.

Spec Kit and Spec2Cloud receive a low rating for setup ease. Spec Kit generates or updates 10 files before implementation, including a constitution, specification, requirements checklist, research documentation, data models, an implementation plan, and a task list. Spec2Cloud generates 16 files, including technology and architecture documentation, a product requirements document, six feature requirements documents, an architecture decision record, and an increment plan. In the comparison, applying either workflow takes approximately two hours before implementation can begin. During this time, the researcher initiates several sequential workflow steps and reviews the artefacts produced at each stage.

AIUP receives a medium rating. It generates 13 files, although nine of these are separate documents for the generated use cases. Its workflow contains fewer sequential steps than Spec Kit and Spec2Cloud. It also requires slightly less time before implementation can begin than for Spec Kit and Spec2Cloud, but it requires manual preparation to adapt the Claude plugin for GitHub Copilot and create the initial documentation required by the workflow.

OpenSpec and Tessl receive a high rating for setup ease and each reaches implementation in approximately 30 minutes. OpenSpec generates five files across four artefact types: a proposal, a design, two specifications, and a task list. It proceeds to implementation after two user-invoked prompts. Tessl generates one main specification file after its requirements-gathering and specification-writing steps. Although Tessl asks clarification questions that require user input, its pre-implementation workflow remains relatively lightweight.

2. Databricks compatibility

Databricks compatibility assesses whether each tool produces artefacts and implementation outputs that align with the Databricks environment used in the synthetic case. This includes the correct use of Databricks concepts, source-data metadata, and compatible implementation technologies.

Spec Kit shows the strongest Databricks compatibility. Its workflow includes a research step that inspects the available source data and generates data models and reporting schemas based on this information. Spec Kit also uses the Databricks command-line interface (CLI), which enables interaction

with a Databricks workspace through a terminal. Through this interface, the programming agent retrieves metadata about the source tables and uses the resulting table and column names in the generated artefacts.

OpenSpec, Tessl, and Spec2Cloud show partial Databricks compatibility. OpenSpec identifies the correct source name in the catalog without performing an explicit source-inspection step, but subsequently includes columns that are not present in the source data. Tessl describes the data product using a medallion architecture, which is commonly used in Databricks environments, but uses an incorrect source name and introduces columns that do not exist. Spec2Cloud generates architecture and data-model artefacts but introduces Azure-specific components, such as Azure Blob Storage, that are not part of the provided case.

AIUP shows limited Databricks compatibility. Although it generates requirements, an entity model, and use-case specifications, its implementation workflow assumes the use of Vaadin and jOOQ. It does not provide a way to switch this implementation workflow to the Databricks and Python technologies used in the synthetic case.

3. Assumption accuracy

Assumption accuracy assesses the extent to which the generated outputs remain consistent with the provided case description, available source data, and user-confirmed decisions. An assumption is considered unsupported when it is neither provided in the case nor confirmed by the user. The assumptions identified during the comparison concern data attributes, architecture, permissions, infrastructure, and implementation technologies.

Spec Kit receives a medium rating. Its research step grounds parts of the generated artefacts in the available source data, but its permission configuration causes errors during implementation. It also adds edge cases that are not included in the case description. These edge cases are not necessarily incorrect, but the workflow does not ask the user to confirm them before including them in the specification.

OpenSpec and Tessl receive low ratings. OpenSpec generates an incorrect permission configuration and includes columns that are not present in the source data. Tessl asks clarification questions about data attributes that do not exist in the source data and subsequently includes unsupported source information in its specification.

Spec2Cloud receives a medium rating. It initially generates its product and feature requirements documents from the existing project template before incorporating the case description. After the case description is provided, the documents are updated, but they continue to include Azure-specific components that are not part of the synthetic case. AIUP also receives a medium rating. Its requirements, entity model, and use cases incorporate the case description, but its implementation workflow assumes the use of Vaadin and jOOQ, which are not part of the Databricks and Python context.

The generated artefacts of Spec Kit, OpenSpec, and Tessl also introduce a medallion architecture without requesting confirmation, even though this architecture is not specified in the case or used as a default in the practice context. In addition, some generated implementation artefacts select a specific Databricks cluster without confirmation. This choice is relevant because the available compute configurations differ in their intended use and cost.

4. Clarification support

Clarification support assesses whether a tool requests additional information or confirmation when the initial request is incomplete or ambiguous. Effective clarification occurs before unsupported assumptions are incorporated into the generated artefacts.

Spec2Cloud receives a high rating. Its workflow contains mandatory review checkpoints, referred to by the tool as human gates, after each phase. At these checkpoints, the user reviews the generated artefacts, confirms proposed decisions, and provides missing information before the workflow continues. This creates explicit opportunities for clarification throughout the workflow. However, the questions are presented in an open-ended format, which requires the user to formulate a free-text response rather than confirm or select from structured options.

Tessl receives a medium rating. Its requirements-gathering step asks clarification questions and requires user confirmation before the specification is generated. However, some questions refer to attributes that

are not included in the case description or available source data. The user must therefore correct these questions instead of only providing missing information.

Spec Kit, OpenSpec, and AIUP receive low ratings. Spec Kit does not include clarification before generating its main artefacts and instead relies on an optional analysis step to identify inconsistencies afterwards. OpenSpec generates its proposal, design, specifications, and task list without explicitly requesting missing information. AIUP likewise does not include a guided clarification activity before generating its requirements and use-case artefacts.

5. Data-model clarity

Data-model clarity assesses how explicitly each tool represents the structure of the data product, including datasets, attributes, relationships, data layers, and schema definitions. This criterion concerns the visibility and level of detail of the representation rather than the correctness of its contents, which is addressed under assumption accuracy.

Spec Kit, AIUP, and Spec2Cloud receive high ratings. Spec Kit generates a data model and reporting schema that describe the data layers, entity relationships, transformations, and reporting tables. AIUP generates an entity model and an entity relationship diagram (ERD), together with descriptions of the tables, attributes, data types, and validation rules. Spec2Cloud generates architecture and data-model documentation containing an ERD and descriptions of the relevant data structures. These artefacts make the structure and relationships of the proposed data product explicit.

OpenSpec and Tessl receive medium ratings. OpenSpec generates separate table-specific specifications but does not provide a consolidated representation of the data structure or the relationships between tables. Tessl describes the proposed data layers and architecture in its specification, but does not generate a detailed schema or visual data model.

6. Progress tracking

Progress tracking assesses whether a tool shows the current workflow stage, completed activities, and remaining work through phases, task lists, status information, or dashboards.

Spec2Cloud receives a high rating. It divides its workflow into explicit phases, such as B0, B1, and B2, and generates task lists aligned with these phases. Its mandatory review checkpoints and audit logs also record completed activities and decisions. Together, these mechanisms allow the user to identify the current workflow stage and the steps that remain.

Spec Kit, OpenSpec, and Tessl receive medium ratings. Spec Kit generates a task list and supports repeated analysis of the generated artefacts, but does not provide a single overview of the current workflow phase. OpenSpec provides a status overview and dashboard for viewing active changes and generated artefacts. However, during the comparison, its task overview does not consistently reflect all completed items. Tessl creates and updates a task list during implementation but does not show progress during its earlier requirements-gathering and specification-writing steps.

AIUP receives a low rating. It does not provide a structured overview of the workflow state or remaining activities. The user must consult the documentation to determine which activity to initiate next.

7. Adaptability

Adaptability assesses how easily a tool can be adjusted to a specific project context. This includes whether its workflow, skills, prompts, and artefact templates can be modified. When no adaptation is performed during the comparison, the rating is based on the number and distribution of workflow components that contain context-specific instructions.

AIUP receives a low rating. It requires substantial manual adaptation before it can be used with GitHub Copilot because it is originally provided as a Claude plugin. In addition, its implementation workflow is predefined for Vaadin and jOOQ and does not allow the user to select a different technology stack, further limiting its adaptability.

Spec2Cloud receives a medium rating. It is not adapted during the comparison, so its rating is based on its workflow structure. The tool contains many predefined phases, skills, and artefact templates, with

context-specific instructions distributed across several components. Adapting the complete workflow would therefore require changes in multiple places.

Spec Kit also receives a medium rating. Its workflow instructions are accessible for modification, but they are distributed across multiple structured steps, templates, and supporting files. OpenSpec receives a high rating because its workflow contains a relatively small number of separately stored skills and artefact templates. This limits the number of files that need to be modified, and its structure appears to support the addition of context-specific skills without replacing the main workflow. Tessl receives a medium rating. Its workflow is relatively lightweight, but changes can affect both its individual skills and their orchestration.

8. Validation support

Validation support assesses whether a tool provides mechanisms for checking the correctness and consistency of generated artefacts and verifying the implementation against the specified requirements.

Spec Kit and Spec2Cloud receive high ratings. Spec Kit includes an analysis step that checks the specification, plan, and task list and reports issues using critical, high, medium, and low severity levels. This allows the user to correct identified issues and repeat the analysis. During the comparison, however, the process does not reach a clear completion state. After the reported issues are addressed, rerunning the analysis produces additional findings because the revised artefacts are assessed again without an explicit condition for declaring the analysis complete.

Spec2Cloud includes what the tool calls a testability gate. This gate does not execute tests but requires the user to determine how the features described in the feature requirements documents (FRDs) can be verified before the workflow continues. Spec2Cloud also requires manual validation activities, including checking the implementation against the FRD items and running the relevant validation procedures.

AIUP, Tessl, and OpenSpec receive low ratings. Neither of these tools provide a structured validation of either the generated artefacts or the implementation.

4.1.3 Design insights

The tool comparison identifies limitations that inform the design of the proposed tool. Table 4.4 presents five general design insights concerning Databricks compatibility, clarification, data-model clarity, and validation. These insights specify the workflow elements that should be added or changed to reduce unsupported assumptions, improve alignment with Databricks, and support the programming agent during specification and implementation.

DI-1 concerns the addition of a Databricks-specific skill and requires further specification because it covers several types of platform-specific information and decisions. Table 4.5 therefore decomposes DI-1 into six subordinate insights, DI-1.1 to DI-1.6. These subordinate insights describe the required functions of the Databricks-specific skill and form part of DI-1 rather than separate top-level design insights.

4.2 Tool selection

Based on the comparison, OpenSpec is selected as the baseline tool for the next design iteration. This decision is not based on OpenSpec having the strongest overall performance, but on its suitability as a foundation for further design development. Compared to Spec-kit and Spec2Cloud, OpenSpec provides a more lightweight workflow with fewer generated artifacts and lower setup effort. At the same time, it still covers the core SDD workflow steps of specification and implementation. This makes it easier to adapt than tools with more extensive predefined phases, skills, and documentation structures.

The main weakness of OpenSpec is its limited alignment with the Databricks context. In the tool comparison, which serves as the baseline evaluation, OpenSpec generates useful specification artifacts but also introduces incorrect assumptions, including columns that do not match the source data. It also provides limited clarification and validation support. However, the comparison suggests that Databricks compatibility can be improved by giving the tool more explicit access to platform-specific context. This insight is derived especially from Spec Kit, which shows stronger Databricks alignment after performing a research step and using Databricks CLI commands.

ID	Design insight	Criterion	Observation
DI-1	Add a Databricks-specific skill before planning.	Databricks compatibility	Generic tools use incorrect tables, columns, or cloud-specific concepts.
DI-2	Add a clarification skill that asks targeted questions when required information is missing.	Assumption accuracy and clarification support	Tools make unsupported assumptions when requirements or context are incomplete.
DI-3	Use interactive GitHub Copilot questions to collect clarifications within the workflow.	Clarification support	Open-ended questions interrupt the workflow and increase user effort.
DI-4	Require an explicit data-model specification for data-product development.	Data-model clarity	Explicit schemas and relationships make the implementation target clearer.
DI-5	Add a validation skill that checks the generated specification before implementation.	Assumption accuracy and validation support	Unsupported assumptions are sometimes detected only during or after implementation.

Table 4.4: General design insights derived from the tool comparison

ID	Insight	Observation
DI-1.1	Inspect catalogs, schemas, tables, and column metadata before generating the specification.	Columns are invented or do not match the source data.
DI-1.2	Retrieve available compute options and require confirmation before selecting a cluster type.	The cluster type is assumed.
DI-1.3	Check whether the implementation should use a Databricks job, pipeline, notebook, or another execution setup.	Tools assume either a pipeline or job structure.
DI-1.4	Treat architecture patterns as design decisions that must be confirmed by the user.	A medallion architecture is introduced without confirmation.
DI-1.5	Check available permission groups and access rights through the Databricks CLI or API.	Permission groups are assumed.
DI-1.6	Validate generated artefacts against Databricks-specific concepts and identify unrelated cloud assumptions.	Unrelated cloud concepts are introduced.

Table 4.5: Further elaboration of DI-1 for the Databricks-specific skill

Therefore, OpenSpec is selected as a suitable baseline tool for further design experimentation. Its lightweight and adaptable structure makes it appropriate for testing whether a Databricks-specific skill can improve compatibility, clarification support, and assumption accuracy.

4.2.1 OpenSpec

This subsection describes how OpenSpec uses prompts and skills, which activities form the main workflow, and which artefacts are created during each activity. This also distinguishes the existing OpenSpec components from the extensions introduced later in this chapter.

In the evaluated GitHub Copilot setup, each OpenSpec activity is represented by both a prompt and a corresponding skill. The prompts are stored in `.github/prompts`, while the skills are stored in `.github/skills`. They do not represent separate workflow activities. Instead, the prompt and corresponding skill provide instructions for the same activity through the file structures recognised by GitHub Copilot. The user initiates an activity through a slash-prefixed prompt in the GitHub Copilot chat, after which the programming agent follows the corresponding OpenSpec instructions. The workflow used as the basis for the proposed tool consists primarily of `/opsx-propose` and `/opsx-apply`. The user invokes `/opsx-propose` together with a description of the requested change. This activity creates a directory for the change under `openspec/changes` and generates `proposal.md`, `design.md`, `tasks.md`, and one or more specification files under `specs/`. The proposal describes the reason and scope of the change, the specifications describe the required behaviour, the design records the technical approach and relevant decisions, and the task list divides the implementation into concrete steps.

After reviewing and, where necessary, modifying these artefacts, the user invokes `/opsx-apply`. This activity reads the proposal, specifications, design, and task list and uses them to guide the implementation. The programming agent creates or modifies the implementation files and records completed tasks in `tasks.md`. The apply activity therefore does not create a new set of specification artefacts but uses the artefacts generated through `/opsx-propose` as its implementation input. Figure 4.1 visualises the interaction between the developer, programming agent, OpenSpec instructions, and project repository during the main OpenSpec workflow. The programming agent loads the prompt and skill instructions associated with each activity. During `/opsx-propose`, it uses these instructions to generate the change artefacts. During `/opsx-apply`, it reads these artefacts and uses them to guide the implementation.

OpenSpec also provides three optional activities. The `/opsx-explore` prompt allows the user and programming agent to examine a request, inspect the project, and consider possible approaches without generating specification or implementation files. The `/opsx-sync` prompt incorporates the specifications of an active change into the main project specifications. Finally, `/opsx-archive` moves a completed change, including its proposal, specifications, design, and task list, to the archive. These activities remain available but are not required for the main proposal-to-implementation workflow used by the proposed tool.

Figure 4.2 presents the resulting OpenSpec project structure. The figure separates the files that provide the workflow from the artefacts generated through that workflow. The `.github` directory contains the prompts and skills used by GitHub Copilot, while `openspec/config.yaml` contains the OpenSpec project configuration. Active change artefacts are stored under `openspec/changes/<change-name>`, and completed changes are moved to the archive. The implementation files remain in the normal project structure outside the OpenSpec directory [21].

4.2.2 Refinement through Databricks-specific skill

To improve OpenSpec's Databricks compatibility, a Databricks-specific skill is created using the skill creator functionality in GitHub Copilot. This design intervention is based on the insight that the tools with stronger Databricks alignment have access to more platform-specific context before or during specification generation. The purpose of the skill is therefore to provide OpenSpec with domain-specific guidance for Databricks-based data product development. The skill includes instructions for inspecting source data, using Databricks CLI commands, identifying relevant platform choices, and grounding generated specifications in the actual Databricks environment.

In the first iteration, OpenSpec does not automatically apply the Databricks skill during the workflow. This indicates that simply adding a domain-specific skill is insufficient. The workflow also needs an explicit mechanism for detecting and invoking relevant skills. Therefore, the Databricks skill is refined

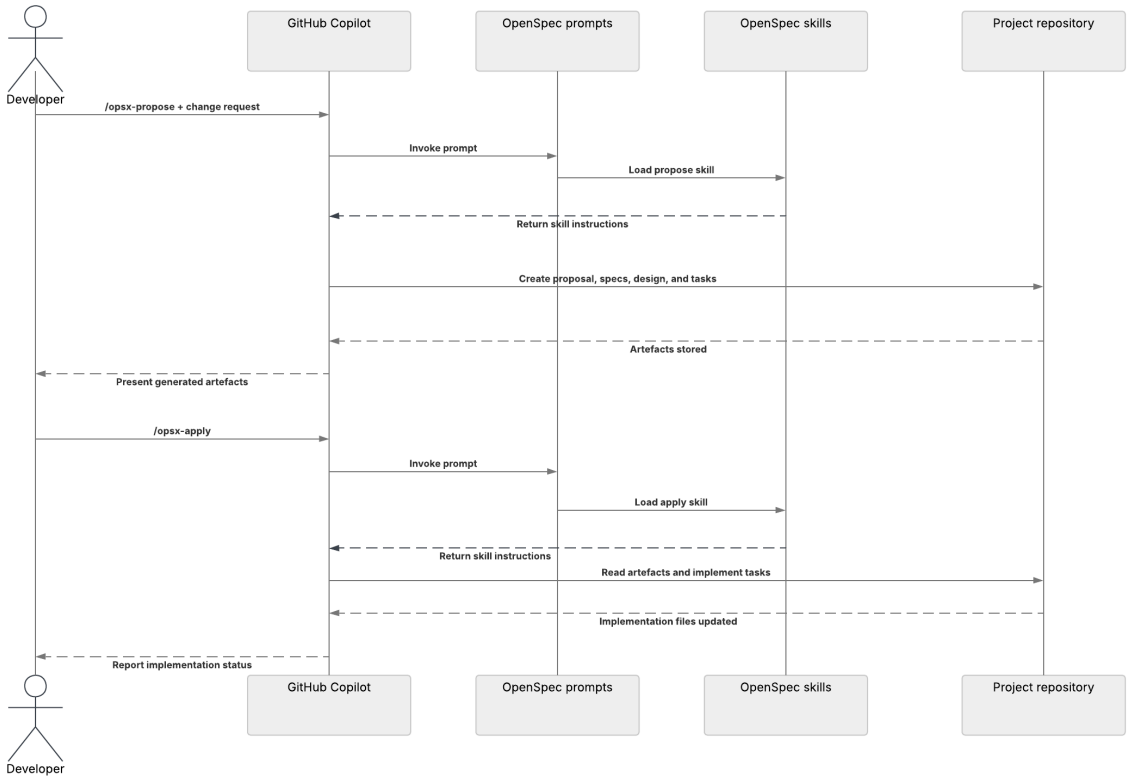


Figure 4.1: UML sequence diagram of the main OpenSpec proposal-to-implementation workflow in GitHub Copilot. OpenSpec prompts and skills are represented as separate lifelines for explanatory purposes, although they are instruction files used by the programming agent rather than independently acting components.

to include steps aligned with the OpenSpec workflow. In addition, the OpenSpec proposal prompt is adapted to check for available domain-specific skills before generating the proposal. The adapted prompt explicitly mentions the Databricks skill as a relevant option and instructs the workflow to follow the steps described in the skill when the use case involves Databricks.

After this refinement, OpenSpec uses the Databricks skill correctly. The workflow asks interactive questions about Databricks-specific design choices, including whether the implementation should use a pipeline or a job, which cluster configuration should be selected, and for which environment the permissions file should be generated. It also uses Databricks CLI commands when additional information about the data environment is required. These changes improve the grounding of the generated artifacts and reduce the number of unsupported assumptions compared to the baseline OpenSpec workflow.

Tool	**2. Databricks compatibility**	**3. Assumption accuracy**	**4. Clarification support**	**5. Data model clarity**	**6. Progress tracking**	**7. Adaptability**	**8. Validation support**
OpenSpec	Medium	Low	Low	Medium	Medium	High	Low
OpenSpec + Databricks skill	High	Medium	High	Medium	Medium	High	Medium

Table 4.6: Comparison of OpenSpec before and after adding the Databricks-specific skill

Table 4.6 compares the baseline OpenSpec workflow with the adapted OpenSpec workflow after adding the Databricks-specific skill. Setup effort is excluded from this comparison because the skill does not substantially change the effort required to run the OpenSpec workflow itself. Instead, the table focuses on the criteria that are most directly affected by the intervention: Databricks compatibility, assumption accuracy, clarification support, data model clarity, progress tracking, adaptability, and validation support.

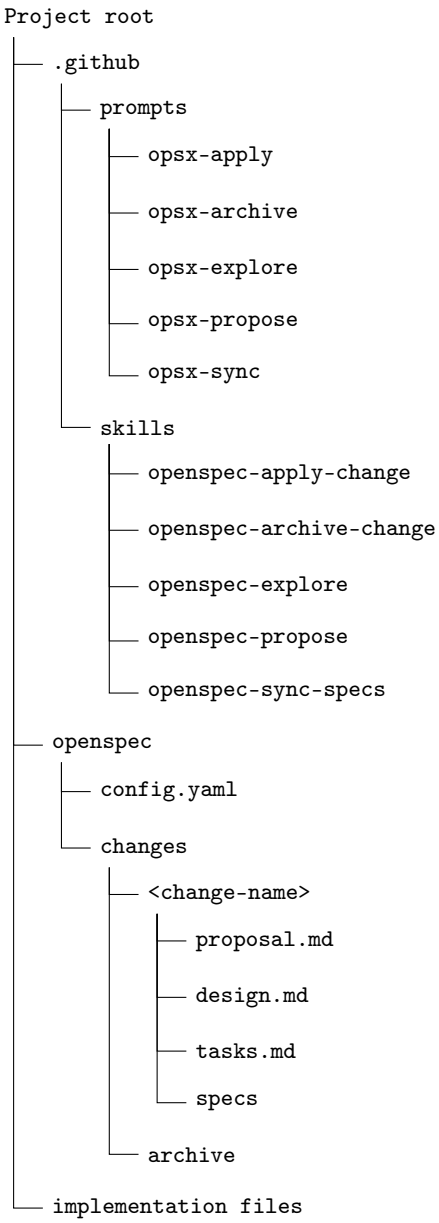


Figure 4.2: Base OpenSpec project structure

Overall, the refinement improves Databricks compatibility, assumption accuracy, clarification support, and validation support, while data model clarity, progress tracking, and adaptability remain unchanged. The Databricks-skill refinement therefore produces the framework-extensibility insight shown in Table 4.7.

1. **Databricks compatibility** Databricks compatibility improves because the adapted workflow uses more platform-specific knowledge during specification and implementation by using Databricks CLI commands and asking Databricks-grounded questions.
2. **Clarification support** Clarification support improves because important Databricks design decisions are moved earlier into the workflow and are presented as structured questions. These questions concern concrete choices such as pipeline or job implementation, cluster configuration, permissions, and environment selection.
3. **Assumption accuracy** Assumption accuracy improves because the adapted workflow gathers more context from the Databricks environment before and during implementation. However, the score remains medium rather than high because the improved workflow does not fully remove the

need for user intervention. During implementation, errors still occur that require the user to choose between alternative solutions. Nevertheless, these choices are more specific to the Databricks context and are presented as structured options rather than open-ended troubleshooting. This suggests that the Databricks skill improves the tool's contextual awareness, but does not fully eliminate unsupported assumptions.

4. **Data model clarity** Data model clarity and progress tracking remain unchanged. Although the Databricks skill improves the contextual grounding of the workflow, it does not introduce a separate data modeling step or additional mechanisms for tracking workflow progress. OpenSpec still relies mainly on its existing specification and task structure.
5. **Adaptability** Adaptability remains high, as the successful integration of the Databricks skill confirms that OpenSpec can be extended with domain-specific functionality without redesigning the entire workflow.
6. **Validation support** Validation support improves from low to medium. The use of Databricks CLI commands provides a limited form of validation because the workflow can retrieve information from the actual Databricks environment instead of relying only on the initial prompt. However, validation support remains medium because the workflow still does not include a structured validation step that systematically checks generated specifications against source schemas, permissions, cluster configuration, and implementation requirements.

ID	Insight	Observation
DI-6	Add an extension mechanism that allows additional skills or workflow steps while preserving the base OpenSpec flow.	The Databricks skill only works after the OpenSpec workflow is adapted to detect and invoke relevant skills.

Table 4.7: Framework extensibility design insight from the Databricks skill refinement

4.3 Software Requirements Specification analysis

This section examines how the OpenSpec artefacts relate to the IEEE 830 SRS structure and which parts of this structure are relevant to data-product development. First, the OpenSpec artefacts are mapped to the SRS sections to identify the requirements information they already cover. Second, the SRS sections are mapped to the technical data-product components identified in Chapter 2. Comparing these mappings identifies the requirements categories that need to be added to or strengthened in the adapted OpenSpec workflow and provides the resulting design insights.

4.3.1 Mapping OpenSpec to SRS sections

After selecting OpenSpec as the basis for the proposed workflow, the next design step is to analyse how its artefacts relate to the structure of a Software Requirements Specification (SRS). This analysis is necessary because the background chapter shows that specifications in Spec-Driven Development can largely be interpreted as natural-language SRS-like artefacts. Therefore, the selected tool is assessed in terms of the extent to which its artefacts support the requirements content expected in an SRS.

OpenSpec structures a change through several Markdown artefacts rather than through one integrated requirements document. Table 4.8 presents the main artefacts and their role in the OpenSpec workflow. This table describes the artefacts in terms of their function within OpenSpec; the comparison with the SRS structure is discussed in the subsequent sections.

The table shows that the proposal.md and specs/ artefacts define the intended change and the required behaviour. In contrast, design.md and tasks.md support subsequent design and implementation activities. This distinction is important when comparing OpenSpec to IEEE 830, because an SRS is concerned with the software product to be developed and should not include detailed design or project planning information.

Table 4.9 shows that the introductory part of the SRS is only partly covered by OpenSpec. The coverage that is present is concentrated in `proposal.md`, which captures the purpose and scope of the proposed change. In particular, the proposal describes the high-level intent of the change and its in-scope and out-of-scope boundaries. This corresponds to §5.1.1 and §5.1.2 of the SRS structure. However, the intended

OpenSpec artefact	Main contents	Role in the OpenSpec workflow
`proposal.md`	Intent, problem/motivation, scope, high-level approach, and expected impact	Describes what should change and why the change is needed
`specs/`	Expected behaviour, scenarios, inputs, outputs, and acceptance criteria	Describes the required behaviour of the product per change
`design.md`	Technical approach, architecture decisions, implementation considerations, alternatives, risks, and trade-offs where relevant	Describes how the proposed change can be realised technically
`tasks.md`	Ordered implementation checklist with checkboxes	Breaks the change down into concrete implementation steps

Table 4.8: OpenSpec artefacts and their roles in the workflow

audience of the SRS, which is also part of §5.1.1, is not explicitly captured in OpenSpec. In addition, OpenSpec does not provide separate sections for definitions, references, or an overview of the document structure. As a result, `proposal.md` can be seen as a partial equivalent of the SRS introduction, but it needs to be extended to fully support this part of the SRS.

SRS section	Required content	OpenSpec artefact	Coverage
§5.1.1 Purpose	Purpose of SRS; intended audience	`proposal.md`	High level intent; intended audience not specified
§5.1.2 Scope	Product name, what it will/won't do, description of application of software	`proposal.md`	Defines the functionality included in and excluded from the proposed change
§5.1.3 Definitions, acronyms, abbreviations	Glossary of all domain terms used in the SRS	—	—
§5.1.4 References	List of referenced documents	—	—
§5.1.5 Overview	Description of SRS contents and how it is organised	—	—

Table 4.9: Coverage of the IEEE 830 SRS Introduction (§5.1) by OpenSpec artefacts [1]

Table 4.10 shows that most elements of the overall description section are not explicitly covered by OpenSpec. Sections such as product perspective, user characteristics, constraints, assumptions and dependencies, and apportioning of requirements do not have a direct corresponding artefact. However, §5.2.2, product functions, is covered by `proposal.md`. The proposal describes the intended change and its scope, which provides a summary of the major functions that the software is expected to perform.

Table 4.11 shows that OpenSpec primarily covers the functional part of the specific requirements section. The `specs/` artefact describes the required behaviour of a capability through requirement statements, typically formulated as "shall" statements. This corresponds closely to §5.3.2, where the SRS specifies the functions that the software must perform. Input and output behaviour is also represented in `specs/`, since capability specifications describe how the system should respond in specific scenarios and what outputs are expected for given inputs. This provides coverage of §5.3.1 at a behavioural level. However, OpenSpec does not require external interfaces to be specified using a fixed structure that includes names, purposes, sources or destinations, ranges, units, timing, and data formats. Therefore, external interfaces are covered only to the extent that they are part of the behavioural specification. The remaining parts of §5.3 are not explicitly covered by the default OpenSpec structure. Performance requirements, logical database requirements, design constraints, and software system attributes such as reliability, availability, security, maintainability, and portability do not have a direct corresponding artefact. This indicates that `specs/` provides a strong basis for functional requirements, but needs to be extended to support the full specific requirements section of an SRS.

Based on the content of the three main parts of an SRS and the observed alignment with the OpenSpec

SRS section	Required content	OpenSpec artefact	Coverage
§5.2.1 Product perspective	System/user/hardware /software/communication interfaces; memory; operations; site adaptation requirements	—	—
§5.2.2 Product functions	Summary of major functions that the software performs	`proposal.md`	Scope describes major functions as summaries.
§5.2.3 User characteristics	General characteristics of intended users, education level, experience, technical expertise	—	—
§5.2.4 Constraints	general description of any items that will limit the developer's options	—	—
§5.2.5 Assumptions and dependencies	Factors that affect requirements if they change	—	—
§5.2.6 Apportioning of requirements	Requirements deferred to future versions	—	—

Table 4.10: Coverage of the IEEE 830 SRS Overall Description (§5.2) by OpenSpec artefacts [1]

artefacts, a distinction can be made between the role of `proposal.md` and the role of `specs/`. The first two parts of the SRS, the introduction and the overall description, mainly provide contextual information about the product to be developed. They describe the purpose, scope, product context, major functions, users, constraints, assumptions, and dependencies. These elements align most closely with `proposal.md`, which defines the proposed change, its rationale, and its scope. The third part of the SRS contains the specific requirements. These requirements describe what the system shall do in a more precise and verifiable form, including functional behaviour, input and output behaviour, interfaces, performance requirements, data requirements, and software system attributes. This aligns most closely with the OpenSpec `specs/` artefact, which specifies the required behaviour of capabilities using requirement statements. Therefore, `proposal.md` and `specs/` can together be interpreted as the SRS-equivalent requirements artefacts. The `proposal.md` artefact represents the introductory and overall-description parts of the SRS, while `specs/` represents the specific requirements part. In this interpretation, `design.md` and `tasks.md` are not part of the SRS-equivalent specification itself. Instead, they are supporting artefacts derived from the requirements captured in `proposal.md` and `specs/`.

Although IEEE 830 does not require an SRS to follow one exact outline, it does describe the types of information that should be present in a good SRS. Therefore, the mapping in this section should not be interpreted as a requirement to reproduce the IEEE 830 structure directly in OpenSpec. Instead, the SRS structure is used as a reference for identifying which SRS parts are already supported by OpenSpec and which are missing.

SRS section	Required content	OpenSpec artefact	Coverage
§5.3.1 External interfaces	All inputs/outputs with name, purpose, source/destination, range, units, timing, data format	`specs/`	Specs may describe I/O behaviourally; no mandatory schema for range, timing, units, or format
§5.3.2 Functions	Functional requirements as “shall” statements; validity checks, sequences, error handling, I/O relationships	`specs/`	Functional behaviour per capability in ”shall” statements
§5.3.3 Performance requirements	Measurable static/dynamic numerical requirements	—	—
§5.3.4 Logical database requirements	frequency of use, accessing capabilities, Data entities and relationships, integrity constraints, data retention requirements	—	—
§5.3.5 Design constraints	Constraints from other standards, hardware limitations etc.	—	—
§5.3.6.1 Reliability	Factors required for reliability at time of delivery	—	—
§5.3.6.2 Availability	Checkpoint, recovery, restart requirements	—	—
§5.3.6.3 Security	Access control, cryptography, integrity check requirements	—	—
§5.3.6.4 Maintainability	Modularity, complexity bounds as requirements	—	—
§5.3.6.5 Portability	Host/OS dependency bounds	—	—

Table 4.11: Coverage of the IEEE 830 SRS Specific Requirements (§5.3) by OpenSpec artefacts [1]

4.3.2 Mapping SRS sections to data product components

Identifying the alignment between OpenSpec and the SRS structure does not yet determine which SRS sections are necessary for data-product development. Therefore, this section maps the SRS sections to the technical data product components identified in Table 2.12. This mapping is used to determine which parts of the SRS should be included or strengthened in the adapted OpenSpec workflow for data products. Table 4.12 presents the resulting mapping. The first column lists the technical data product components, the second column identifies the relevant SRS sections, and the third column describes the coverage of each section in the context of data-product development.

Component	SRS section	Coverage
Data	§5.3.1 External interfaces	Data inputs and outputs
	§5.3.4 Logical database requirements	Data entities, relationships, integrity, and retention
Metadata	§5.1.3 Definitions, acronyms, and abbreviations	Business and semantic definitions
	§5.1.4 References	Sources of metadata definitions
	§5.3.1 External interfaces	Metadata inputs and outputs
	§5.3.4 Logical database requirements	Metadata entities and relationships
Code	§5.2.1.4 Software interfaces	External software dependencies

	§5.3.2 Functions	Required pipeline or application behaviour
	§5.3.5 Design constraints	Implementation constraints and standards
Transformation Logic	§5.3.1 External interfaces	Transformation inputs and outputs
	§5.3.2 Functions	Transformation rules and processing behaviour
	§5.3.3 Performance requirements	Processing speed and throughput
Infrastructure	§5.2.1.4 Software interfaces	Platform and tooling dependencies
	§5.2.1.6 Memory constraints	Storage or memory limitations
	§5.3.3 Performance requirements	Capacity, scalability, and response time
	§5.3.5 Design constraints	Platform or environment constraints
Policies	§5.1.4 References	Referenced policy
	§5.2.4 Constraints	Regulatory and governance constraints
	§5.3.2 Functions	Policy enforcement behaviour
	§5.3.5.1 Standards compliance	Required governance standards
Data Quality	§5.3.2 Functions	Validation and error handling
	§5.3.3 Performance requirements	Measurable quality thresholds
	§5.3.5.1 Standards compliance	Required quality standards
	§5.3.6.1 Reliability	Correctness and reliability of delivered data
SLA/SLO Management	§5.3.3 Performance requirements	Freshness, latency, and throughput targets
	§5.3.6.1 Reliability	Reliability commitments
	§5.3.6.2 Availability	Availability, recovery, and restart requirements
Identity & Access Mechanism	§5.2.1.4 Software interfaces	Identity provider or access-management dependencies
	§5.2.1.5 Communications interfaces	Interfaces to communications, such as local network protocols.
	§5.3.1 External interfaces	Authentication or authorization interfaces
	§5.3.2 Functions	Authentication and authorization behaviour
	§5.3.6.3 Security	Access restrictions and security controls
Change & Incident Management	§5.3.2 Functions	Error handling and recovery behaviour
	§5.3.6.2 Availability	Incident recovery and restart requirements

Table 4.12: Mapping of data product components to SRS sections and required coverage

Table 4.12 shows that data-product components are covered by different types of SRS sections. The Data and Metadata components mainly require interface and logical database requirements, since they concern inputs, outputs, structures, definitions, and relationships. Code and transformation logic are more closely related to functional requirements, software dependencies, and performance requirements. The table also shows that operational components, such as Infrastructure, Policies, Data Quality, SLA/SLO Management, and Identity and Access Mechanisms, require requirements beyond functionality. These components are linked to constraints, standards compliance, reliability, availability, security, and performance. Overall, the mapping indicates that data-product development mainly depends on the specific requirements part of the SRS, especially external interfaces, functions, performance requirements, logical database requirements, and software system attributes. However, some contextual sections, such as definitions, references, constraints, and software interfaces, are also needed to specify the product clearly.

Table 4.13 compares the SRS sections covered by the OpenSpec artefacts with the SRS sections that are relevant for specifying the identified data product components. The table shows that OpenSpec already covers some SRS sections that are also relevant for data-product development, especially §5.3.1 External interfaces and §5.3.2 Functions. These sections are therefore retained in the adapted workflow, but need to be made more explicit for data products. The table also shows several SRS sections that are relevant for data-product development but are not structurally covered by OpenSpec. These include definitions, references, product perspective, constraints, performance requirements, logical database requirements, design constraints, reliability, availability, and security. These sections represent the main gaps between

	§5.1.1	§5.1.2	§5.1.3	§5.1.4	§5.1.5	§5.2.1	§5.2.2	§5.2.3	§5.2.4	§5.2.5	§5.2.6	§5.3.1	§5.3.2	§5.3.3	§5.3.4	§5.3.5	§5.3.6.1	§5.3.6.2	§5.3.6.3	§5.3.6.4	§5.3.6.5
OpenSpec coverage	x	x					x					x	x								
Data product coverage			x	x		x			x			x	x	x	x	x	x	x	x		

Table 4.13: Overview of OpenSpec coverage and data-product relevance across IEEE 830 SRS sections

the OpenSpec structure and the sections needed for data products. In the adapted workflow, these gaps should therefore be addressed by extending either `proposal.md` or `specs/`.

Some SRS sections are covered by OpenSpec but are not directly mapped to a technical data product component. This applies to §5.1.1 Purpose, §5.1.2 Scope, and §5.2.2 Product functions. These sections should not be excluded from the design, because they provide the general context and boundaries of the product to be developed. Instead, they remain part of `proposal.md`, where they describe the purpose, scope, and high-level functions of the data product. The sections that are not covered by OpenSpec and are not directly mapped to a data product component are not included as mandatory parts of the adapted workflow. This applies to §5.1.5 Overview, §5.2.3 User characteristics, §5.2.5 Assumptions and dependencies, §5.3.6.4 Maintainability, and §5.3.6.5 Portability. These sections may still be relevant in specific cases, but the mapping does not identify them as core requirements for the data products considered in this thesis. They can therefore be treated as optional or left implicit in the structure of the workflow.

ID	Insight	Observation
DI-7	Extend `proposal.md` with missing contextual SRS sections.	Definitions, references, product perspective, and constraints are relevant for data products but not structurally covered by OpenSpec.
DI-8	Extend `specs/` with data-product-specific requirement categories.	Performance requirements, logical database requirements, design constraints, reliability, availability, and security are relevant for data products but not structurally covered by OpenSpec.
DI-9	Make data inputs, outputs, and logical data requirements explicit.	Data and metadata require specification of interfaces, entities, relationships, integrity, retention, and semantic definitions.

Table 4.14: Design insights from the SRS and data-product mapping

The gaps identified through the two mappings are translated into the three design insights presented in Table 4.14. DI-7 concerns the contextual information added to `proposal.md`, while DI-8 concerns the requirement categories added to `specs/`. DI-9 further specifies that data inputs, outputs, entities, relationships, and other logical data requirements must be made explicit. These insights guide the extensions to the OpenSpec artefacts introduced in the following section.

4.4 Tool Design

The tool comparison, the Databricks-specific skill refinement, and the mappings between OpenSpec, IEEE 830, and the technical data-product components produce a set of design insights. In the proposed design, these insights are treated collectively as the design requirements: they specify the capabilities that the tool architecture must provide. This section translates these requirements into architectural components while preserving the baseline OpenSpec workflow. The resulting architecture consists of extension skills, domain skills, and the `configure-openspec` component. Figure 4.3 presents these components and their relationship to the OpenSpec workflow.

The architecture is centered on the existing OpenSpec workflow, which remains responsible for generating the workflow artefacts and performing the implementation. The workflow uses the `/opsx-propose` and `/opsx-apply` prompts. The `/opsx-propose` prompt creates `proposal.md`, `design.md`, `tasks.md`, and one or more specification files under `specs/`. These artefacts record the proposed change, required

behaviour, technical design, and implementation tasks. After the artefacts are reviewed, the `/opsx-apply` prompt uses them to guide the implementation. The additional components extend this workflow at specific points without replacing its existing structure.

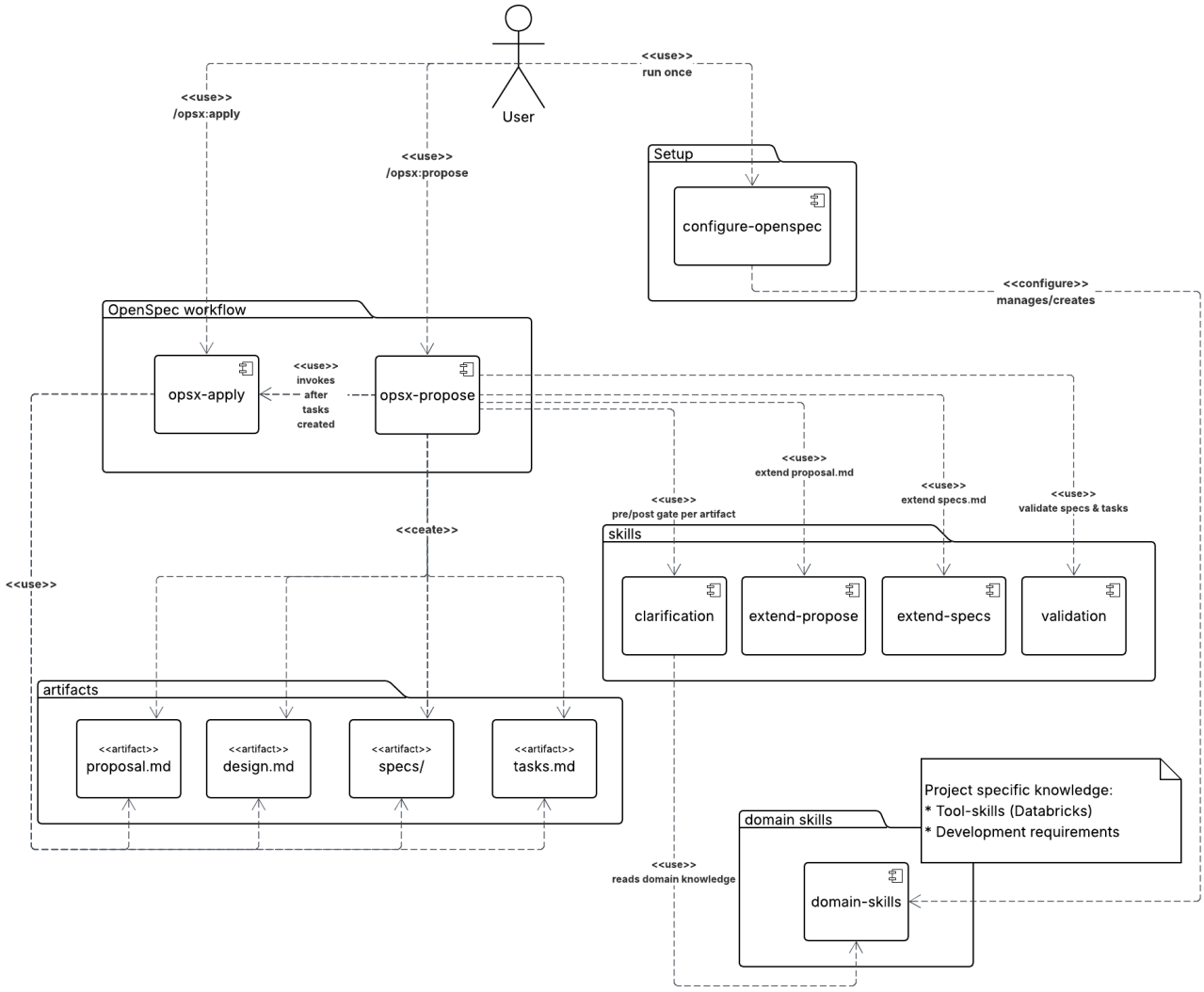


Figure 4.3: Component diagram of tool design including basic OpenSpec workflow

4.4.1 Extension skills

To address the limitations identified during the tool comparison, four additional skills are introduced: `clarification`, `extend-propose`, `extend-specs`, and `validation`. These skills are invoked by the OpenSpec workflow before or after artefact generation and extend the default behaviour without replacing the underlying workflow.

clarification

The clarification skill is responsible for determining whether sufficient information is available before artefacts are generated. When required information is missing, the skill asks targeted clarification questions through GitHub Copilot and collects the additional information before continuing the workflow. In addition, the clarification skill determines whether domain-specific knowledge is required and invokes relevant domain skills when applicable. This component implements design insights DI-2 and DI-3, which address the unsupported assumptions and workflow interruptions observed during the tool comparison.

extend-propose

The extend-propose skill enriches the generated `proposal.md` artefact with the contextual IEEE 830 sections that were identified as missing from OpenSpec but required to cover context for data products. These additions include definitions, references, product perspective, and constraints. As a result, the proposal artefact provides a more complete description of the context surrounding the requested change. This component implements design insight DI-7.

extend-specs

The extend-specs skill enriches the generated `specs` artefact with the SRS sections that were identified as relevant for data-product development but not explicitly covered by OpenSpec. These additions include performance requirements, logical database requirements, design constraints, reliability requirements, availability requirements, and security requirements. The skill also makes data inputs, outputs, entities, relationships, and integrity constraints explicit. Consequently, this component implements design insights DI-4, DI-8, and DI-9.

validation

The validation skill introduces structured validation support before implementation. After specification generation, the skill validates whether requirements can be traced back to the original request, clarification responses, or domain-specific knowledge. It also identifies unsupported assumptions that have been introduced during specification generation. After task generation, the skill verifies whether implementation tasks can be traced back to one or more requirements. This component implements design insight DI-5 and improves traceability throughout the workflow.

4.4.2 Domain skills

Domain skills are managed through the configure-openspec component, meaning they only exist when created through the configure-openspec skill. They can be invoked by the clarification skill whenever additional context is required during the workflow. The repository is organised into two categories: tool skills and development requirements. This structure allows the workflow to combine domain-specific knowledge with project-specific constraints while remaining extensible.

Tool skills

Tool skills provide knowledge and guidance for specific platforms, technologies, or development environments. The proposed tool includes a Databricks skill as a predefined tool skill. This skill was derived from design insight DI-1 and the associated sub-insights DI-1.1 through DI-1.6. The Databricks skill supports the workflow by inspecting catalogues, schemas, tables, and column metadata before specifications are generated. It retrieves available compute options and requires confirmation before a cluster type is selected. Furthermore, the skill determines whether the implementation should use a Databricks job, pipeline, notebook, or another execution mechanism. Architectural patterns, such as the medallion architecture, are treated as explicit design decisions that require user confirmation rather than being assumed automatically. The skill also validates permission groups and access rights through Databricks CLI commands and validates generated artefacts against Databricks-specific concepts to prevent unsupported cloud-specific assumptions. Although Databricks is included as a predefined tool skill, the architecture allows additional tool skills to be added for other technologies and platforms. These skills can be created and maintained through the configure-openspec component without requiring modifications to the core workflow.

Development Requirements

Development requirements represent project-specific or organisation-specific constraints that should be applied throughout the workflow. Examples include coding standards, naming conventions, architectural principles, deployment requirements, security policies, and documentation standards. Unlike tool skills, which are only invoked when relevant to the implementation context, development requirements are automatically applied whenever they are available. This ensures that generated artefacts remain aligned with organisational standards regardless of the underlying technology or platform. This type of skill is

not automatically available; however, they can be created and maintained through the configure-openspec component.

4.4.3 configure-openspec

Unlike the other components, `configure-openspec` is not part of the normal execution workflow. Instead, it acts as a setup mechanism that modifies the OpenSpec workflow so that extension skills and domain skills become available during the execution of `opsx-propose` and `opsx-apply`. The component is also responsible for creating, updating, and managing domain skills. The introduction of this component was motivated by the Databricks skill refinement, where it was observed that simply creating a domain skill was insufficient. The workflow also required a mechanism for discovering and invoking relevant skills. Therefore, `configure-openspec` implements design insight DI-6 by providing an extension mechanism that allows additional domain-specific knowledge to be used without modifying the underlying OpenSpec architecture.

Figure 4.4 presents how the `configure-openspec` skill configures the extension for an OpenSpec project. The skill first inspects the existing OpenSpec configuration and updates the `openspec-propose` skill with the extension workflow. It then asks whether a domain skill needs to be added or updated. Depending on the user's response, it creates a development-requirements skill based on project or organisational requirements, creates a tool-specific domain skill based on relevant documentation, or updates an existing domain skill. After each addition or update, the skill asks whether another domain skill requires attention. This continues until the user indicates that no further skills are needed, after which the completed configuration is reported. During subsequent proposal generation, the clarification skill inspects the available domain skills and identifies those relevant to the current request.

Table 4.15 summarises how the design requirements are implemented by the architectural components. The mapping provides traceability from the findings of the preceding analyses to the resulting tool design and shows that each design insight is addressed by at least one component.

Design Insight	Architectural Component
DI-1 (DI-1.1–DI-1.6)	Databricks skill
DI-2	Clarification skill
DI-3	Clarification skill
DI-4	Extend-specs skill
DI-5	Validation skill
DI-6	Configure-openspec
DI-7	Extend-propose skill
DI-8	Extend-specs skill
DI-9	Extend-specs skill

Table 4.15: Mapping between design insights and architectural components

4.5 Tool Development

This section describes how the architecture presented in Section 4.4 is translated into an implemented extension of OpenSpec. The architectural components are implemented as GitHub Copilot skills using the native `/skill-creator` functionality. For each skill, a description specifies its purpose, when it should be invoked, which artefacts it should read or modify, and which outputs it should produce. This approach translates the components shown in Figure 4.3 into executable workflow extensions while maintaining compatibility with the existing OpenSpec workflow.

The extension skills are implemented first. The `clarification`, `extend-propose`, `extend-specs`, and `validation` skills are created through `/skill-creator` using descriptions derived from the design requirements and architectural design. For `extend-propose` and `extend-specs`, the IEEE 830 Software Requirements Specification standard and the relevant sections identified through the SRS mapping provide additional context during skill creation. This enables the generated skills to extend the OpenSpec artefacts according to the selected SRS sections.

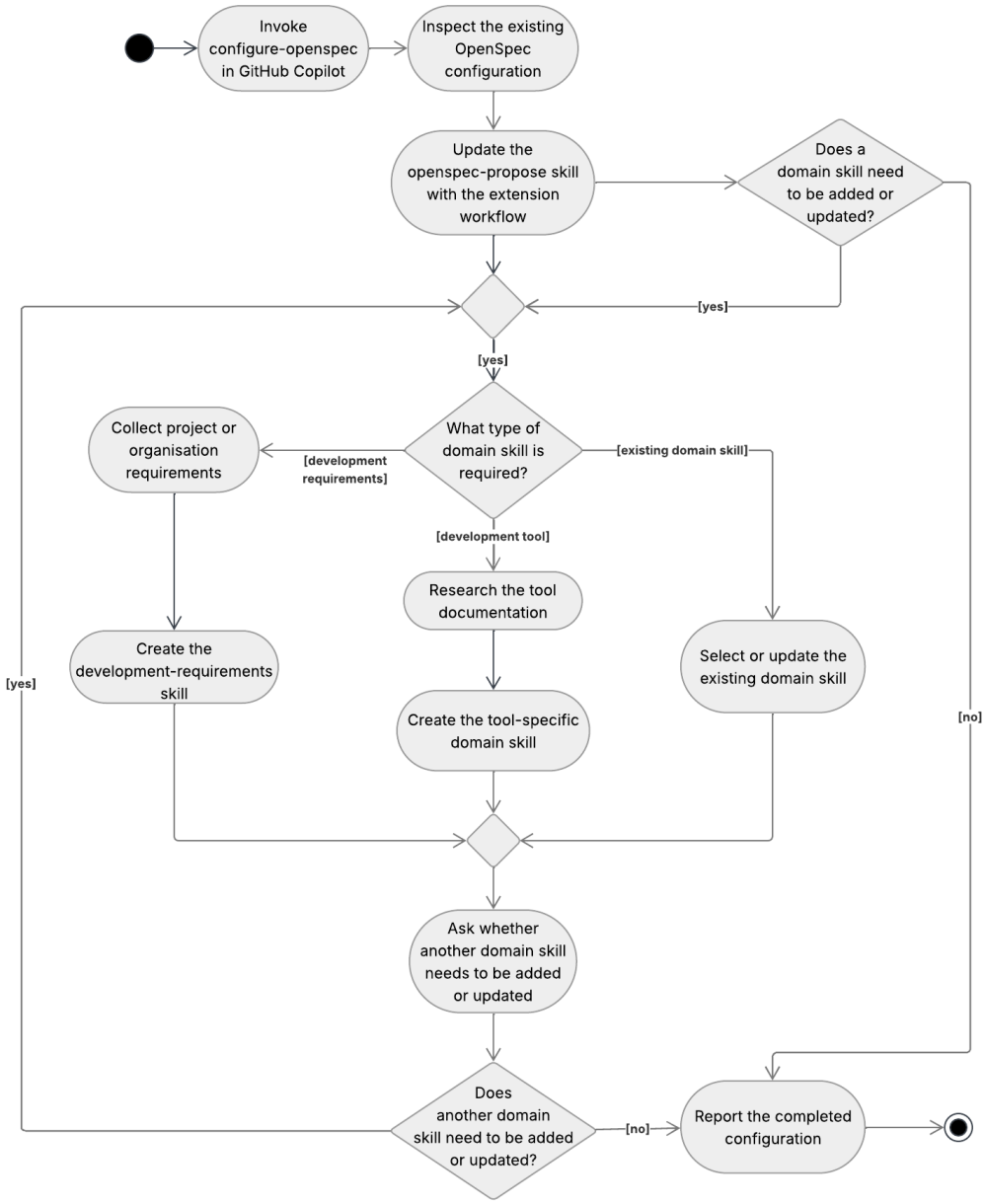


Figure 4.4: UML activity diagram of the `configure-openspec` workflow.

Figure 4.5 shows part of the implemented `extend-specs` skill. The skill definition specifies when it is invoked, which inputs it receives, and which additional requirements categories it adds to the OpenSpec specification. These categories correspond with the gaps identified through the SRS analysis, including performance requirements, logical database requirements, design constraints, reliability, availability, and security.

The `configure-openspec` skill is implemented as the central configuration mechanism of the architecture. It adapts the OpenSpec workflow so that the extension skills and domain skills can be invoked during `/opsx-propose` and `/opsx-apply`. The skill is also responsible for creating and maintaining tool skills and development requirements. When a new tool skill is required, `configure-openspec` performs web research to gather technology-specific information and combines the retrieved information with `/skill-creator` to generate the corresponding skill definition.

This process is used to create the Databricks skill, which serves as the initial tool skill included in the

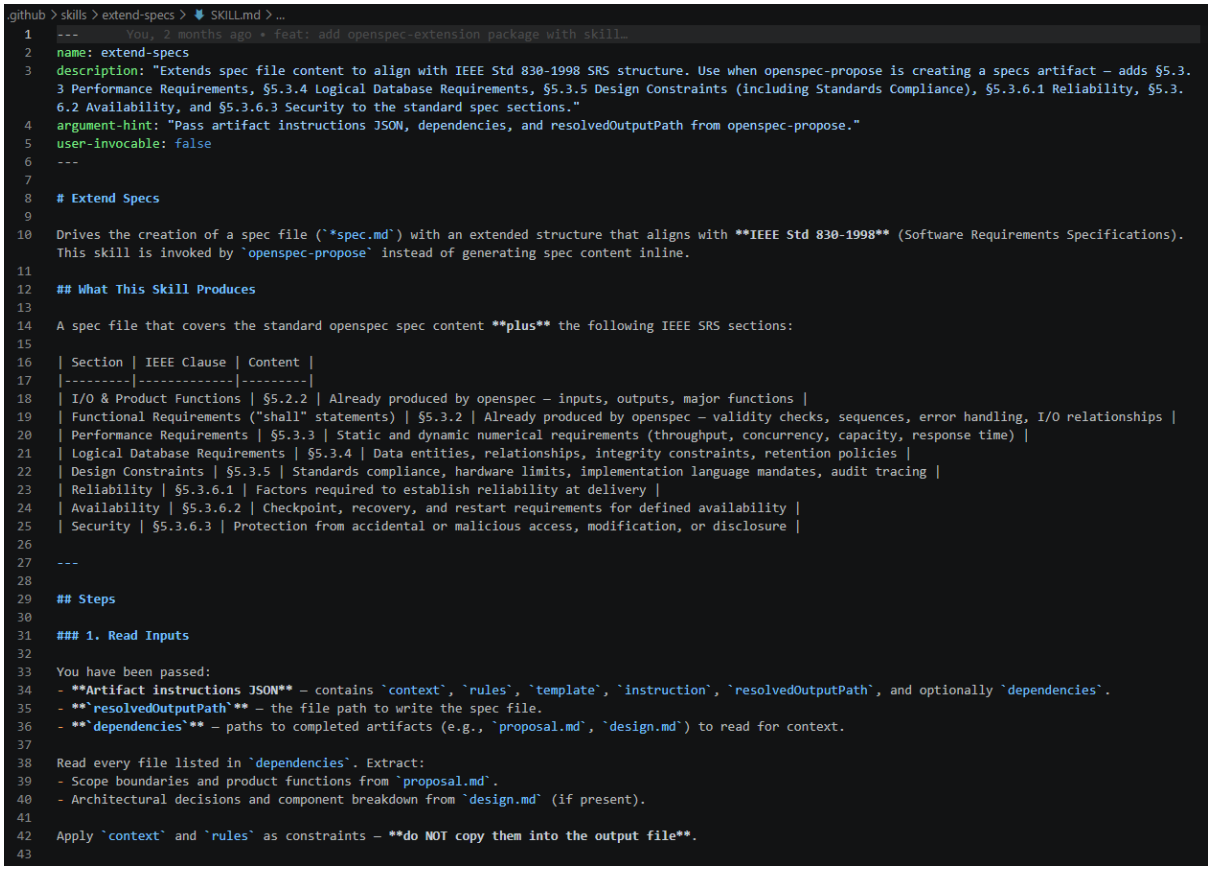


Figure 4.5: Fragment from the implemented `extend-specs` skill showing the additional IEEE 830 requirements categories.

proposed tool. Its definition is based on the Databricks-specific design requirements DI-1.1 through DI-1.6. Development requirements are also managed through `configure-openspec`. Unlike tool skills, these are created from project-specific or organisation-specific requirements supplied by the user. The resulting `development-requirements` skill can subsequently be applied throughout the workflow to align generated artefacts with organisational standards and constraints.

Figure 4.6 shows the resulting implementation structure. The OpenSpec workflow prompts are stored under `.github/prompts`, while the existing and added skills are stored under `.github/skills`. The extension introduces the four extension skills, `configure-openspec`, and the `domain-skills` structure. Active OpenSpec changes and their generated artefacts are stored under `openspec/changes`, while the implementation files remain within the Declarative Automation Bundle template. The figure therefore distinguishes between the files that define the workflow, the artefacts generated by OpenSpec, and the files that form the resulting implementation.

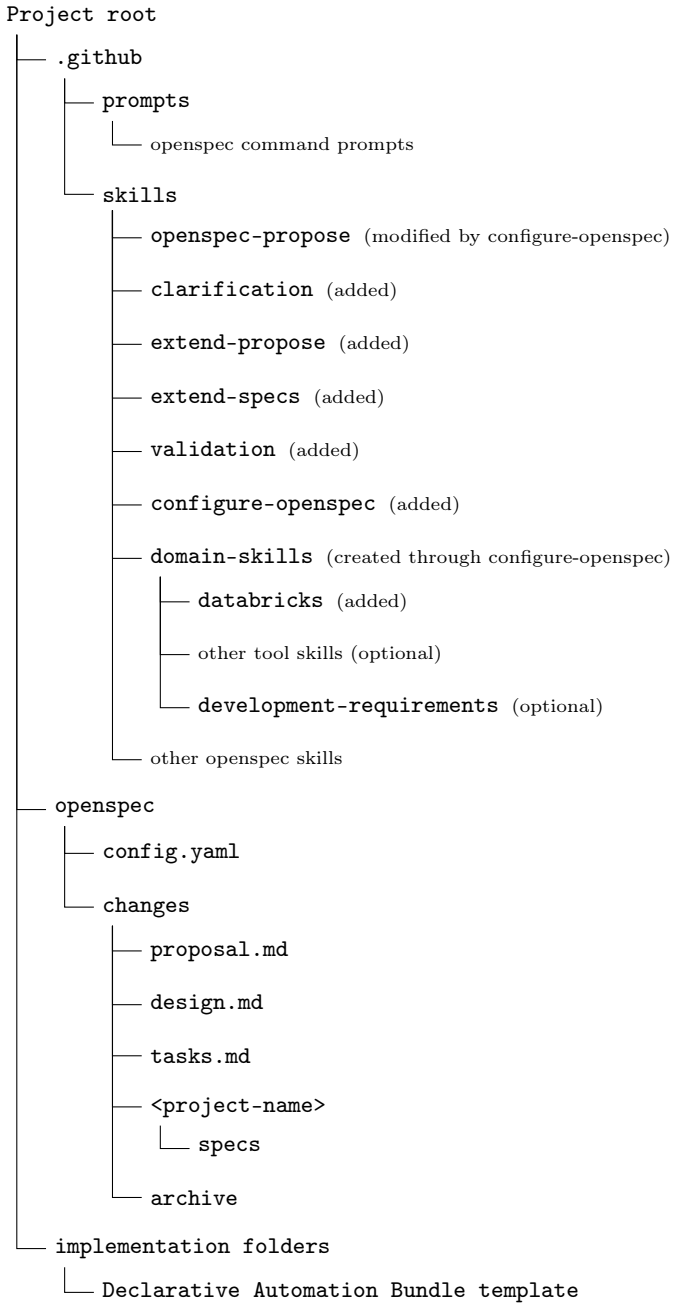


Figure 4.6: Implementation structure of the proposed OpenSpec extension

5 Evaluation

This chapter presents the evaluation of the proposed SDD tool through three successive stages. First, the tool is demonstrated using the synthetic data-product case to examine its operation, identify workflow issues, and verify the resulting refinements. Second, a controlled evaluation with four practitioners examines how participants use and assess the tool when applying it to the same organisational case. The controlled evaluation combines think-aloud observations with a Technology Acceptance Model questionnaire and informs further refinements. Finally, a practical evaluation examines how two participants apply the refined tool to their own ongoing development tasks and compares their questionnaire responses with those from the controlled evaluation.

5.1 Demonstration

This section demonstrates whether the proposed SDD tool operates according to its design before it is evaluated by practitioners. It first describes the case, technical setup, and version of the tool used for the demonstration. It then presents the observations from the initial execution and the refinements made in response. After these refinements are implemented, the demonstration is repeated to verify that the identified issues are resolved. Finally, the refined tool is compared with baseline OpenSpec and OpenSpec with the Databricks skill to show how the successive design iterations differ.

5.1.1 Demonstration setup

The demonstration applies the proposed artefact to the same synthetic data-product case used during the tool comparison. The same case description, presented in Appendix E.1, is provided as the initial request. The demonstration also starts in a copy of the same Declarative Automation Bundle (DAB) template used for each tool in the comparison. The initial request, source data, and project structure therefore remain consistent with the earlier comparison.

The demonstration uses the proposed SDD tool described in Chapter 4. The tool is implemented as an extension of OpenSpec that preserves its existing workflow while adding support for clarification, specification extension, validation, configuration, and domain-specific knowledge. The demonstrated version includes the clarification, extend-propose, extend-specs, validation, and configure-openspec skills, together with the predefined Databricks skill. This configuration represents the complete version produced during the design and development stage and serves as the starting point for the demonstration.

The objective of the demonstration is to assess whether the proposed tool integrates clarification, domain-specific knowledge, SRS-based extensions, and validation into the OpenSpec workflow while generating the required artefacts. During its execution, observations are recorded to identify limitations in the implemented workflow and opportunities for further refinement.

5.1.2 Observations and refinements

The initial demonstration completes the extended OpenSpec workflow and generates the expected artefacts. The clarification skill is invoked during requirements elicitation, the Databricks skill provides domain-specific guidance, and the proposal and specification artefacts are enriched through the extension skills. This shows that the components can be integrated into an executable OpenSpec workflow.

The execution also identifies five technical and orchestration issues in the initial version. These issues concern the sequencing of skills, the timing of validation, opportunities for user review, the handling of implementation issues, and the transition to implementation. They therefore relate to the general operation of the workflow rather than to the content of the synthetic case itself. Each observation is translated into a refinement of the workflow. Table 5.1 summarises the observations and the corresponding refinements.

Observation	Refinement
The `extend-specs` skill was executed after the initial specification had already been generated.	The `configure-openspec` skill was updated to modify the `opsx-propose` prompt so that specification extensions are incorporated during specification generation, instead of only the `openspec-propose` skill.
The `validation` skill was executed before the specification extension was completed.	The `configure-openspec` skill was updated to modify the `opsx-propose` prompt so that validation is executed after specification extensions have been applied, instead of only the `openspec-propose` skill.
The workflow automatically continued after generating an artefact without allowing user review.	The `clarification` skill was extended with post-artefact interactive questions and `configure-openspec` was updated to invoke the clarification skill after artefact generation.
Implementation errors interrupted the workflow and required manual intervention.	The `configure-openspec` skill was updated to modify both `openspec-apply` and `opsx-apply` so that implementation issues are handled through interactive GitHub Copilot questions.
The transition to the `opsx-apply` required a manual restart of the workflow.	The `configure-openspec` skill was updated to modify `opsx-propose` so that an interactive confirmation question is presented after artefact generation and the `opsx-apply` workflow can be invoked directly.

Table 5.1: Observations and resulting refinements from the demonstration

The first two observations concern the interaction between the `extend-specs` and `validation` skills and the OpenSpec specification-generation process. In the initial demonstration, the default OpenSpec process first generates the specification and subsequently extends it with the additional SRS sections. Consequently, validation operates on an intermediate version rather than on the final specification. The refinement is implemented through the `configure-openspec` skill rather than through changes to the individual extension skills. The refined skill modifies both the `openspec-propose` skill and the `opsx-propose` prompt so that specification extensions are incorporated during generation and validation executes only after the final specification is complete.

The third observation concerns the absence of user-review opportunities between generated artefacts. In the initial demonstration, the workflow continues after generating an artefact, which limits the user's ability to review and refine the output before subsequent artefacts are created. The refined `clarification` skill therefore includes post-artefact interactive questions. In addition, `configure-openspec` invokes the clarification skill after each artefact-generation step. This allows the user to continue the workflow or request modifications before proceeding to the next artefact.

The final two observations concern the implementation phase. Implementation errors interrupt the initial workflow and require manual intervention, while the transition from artefact generation to implementation requires the user to initiate the apply phase separately. The refined `configure-openspec` skill modifies both the `openspec-apply` skill and the `opsx-apply` prompt so that implementation issues are presented through interactive GitHub Copilot questions. The `opsx-propose` prompt also presents a confirmation question to continue to implementation after the final artefact generation. When the user confirms the transition, the prompt invokes the apply workflow directly.

5.1.3 Refined demonstration

After the refinements are implemented, the demonstration is repeated using the same synthetic case description. During this repeated demonstration, the specification extensions form part of the specification-generation process, validation executes on the final extended specification, and interactive review checkpoints appear after artefact generation. Implementation-related issues are handled through interactive GitHub Copilot questions, and the confirmation step provides a direct transition from artefact generation to implementation.

Within this repeated application, the five issues identified during the initial demonstration do not recur,

and no additional orchestration issues are observed. The repeated demonstration therefore shows that the refinements resolve the identified issues for the examined synthetic case and workflow path. It does not establish that the tool is free of limitations across all possible case descriptions, because different requirements or project contexts may reveal other information gaps. The resulting version consequently serves as the input to the controlled evaluation, which applies the tool to a different organisational case. The later practical evaluation further examines the refined tool in two ongoing development tasks.

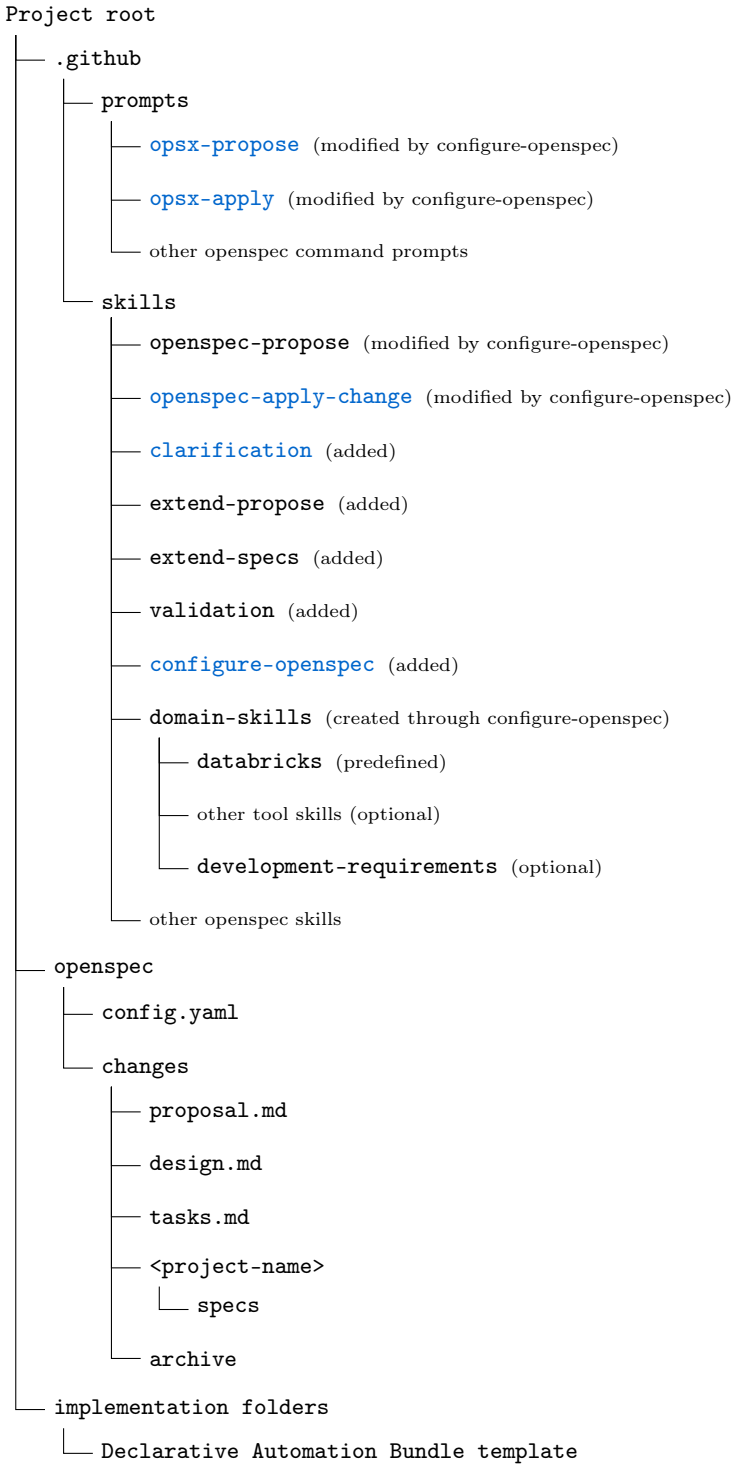


Figure 5.1: Implementation structure of the proposed OpenSpec extension after demonstration refinements

Figure 5.1 presents the implementation structure after these demonstration refinements. The components

highlighted in blue are those affected by the refinements. The clarification and configure-openspec skills are updated, while the opsx-propose and opsx-apply prompts and the openspec-apply-change skill are modified to support the revised sequencing, review checkpoints, handling of implementation issues, and direct transition to implementation. The remaining components retain the structure established during the design and development stage.

Comparison with baseline OpenSpec

After the refined demonstration, the resulting version is compared with baseline OpenSpec and OpenSpec with the Databricks skill resulting from the tool-selection phase. The comparison uses the same criteria as the earlier tool comparison, enabling a direct comparison between the baseline tool, the Databricks-enhanced version, and the proposed SDD tool. Because the comparison uses the same synthetic case, its assessments describe the support observed within this application.

Tool	**2. Databricks compatibility**	**3. Assumption accuracy**	**4. Clarification support**	**5. Data model clarity**	**6. Progress tracking**	**7. Adaptability**	**8. Validation support**
OpenSpec	Medium	Low	Low	Medium	Medium	High	Low
OpenSpec + Databricks skill	High	Medium	High	Medium	Medium	High	Medium
Proposed OpenSpec Extension	High	High	High	High	High	High	High

Table 5.2: Comparison of OpenSpec, OpenSpec with Databricks skill, and the proposed OpenSpec extension

Table 5.2 compares the three configurations in terms of Databricks compatibility, assumption accuracy, clarification support, data-model clarity, progress tracking, adaptability, and validation support. Within the synthetic case, the proposed SDD tool receives higher assessments for assumption accuracy, data-model clarity, progress tracking, and validation support than the preceding configurations. It also retains the Databricks compatibility and clarification support introduced by the Databricks skill.

1. **Databricks compatibility** Databricks compatibility remains high. The proposed tool retains the Databricks skill and therefore uses Databricks-specific knowledge during specification generation and implementation. The workflow can retrieve contextual information through Databricks CLI operations and guide users through Databricks-specific design decisions.

2. **Assumption accuracy** Assumption accuracy receives a high assessment within the synthetic case. Databricks-specific context gathering, clarification questions, interactive review checkpoints, and validation make proposed assumptions visible and provide opportunities for user confirmation before they are incorporated into subsequent artefacts. User input nevertheless remains necessary for case-specific requirements and design decisions.

3. **Clarification support** Clarification support remains high and is extended through post-artefact review checkpoints. In addition to asking clarification questions before artefact generation, the workflow invokes the clarification skill after each generated artefact. This allows users to review the output, request modifications, and provide additional context before the workflow continues.

4. **Data-model clarity** Data-model clarity receives a high assessment. The `extend-specs` skill adds the relevant IEEE 830 sections and makes data-product information more explicit. Inputs, outputs, entities, relationships, and integrity constraints consequently receive explicit locations in the generated specifications.

5. **Progress tracking** Progress tracking receives a high assessment because the workflow includes interactive checkpoints after artefact generation and before implementation. These checkpoints provide explicit transitions between workflow stages and allow users to control whether the workflow proceeds.

6. **Adaptability** Adaptability remains high. The integration of the Databricks skill, extension skills, and `configure-openspec` mechanism indicates that OpenSpec can be extended without replacing

its underlying workflow. The architecture also supports additional tool skills and organisation-specific development requirements.

7. **Validation support** Validation support receives a high assessment. The proposed tool includes a dedicated validation skill that performs structured validation of generated specifications and implementation tasks. Requirements are validated against the initial request and collected context, while implementation tasks are validated against the resulting specifications. This provides more systematic validation than the environmental checks available through the Databricks skill alone.

Within the examined synthetic case, the comparison shows that the Databricks skill primarily adds contextual grounding for the Databricks platform, while the proposed SDD tool adds broader support for requirements elicitation, specification structure, validation, and workflow control. According to the applied comparison criteria, the proposed tool therefore provides more extensive support for data-product development than baseline OpenSpec and OpenSpec with the Databricks skill. Its operation with other case descriptions is examined through the subsequent controlled and practical evaluations.

5.2 Controlled evaluation

Following the technical demonstration, the proposed tool is evaluated with practitioners from the case organisation. Whereas the demonstration examines the operation of the tool using a synthetic taxi-reporting case, the controlled evaluation applies it to an organisational data-product case. The same case and technical setup are used in every session so that participant experiences and observations can be compared.

5.2.1 Controlled evaluation setup

The controlled evaluation consists of four individual sessions, with one session per participant. Each session lasts approximately one hour. The participants are described in Table 3.1.

The demonstration and controlled evaluation use different case descriptions. The demonstration uses the synthetic data-product case presented in Appendix E.1, whereas the controlled evaluation uses a case based on an organisational data-product scenario and source information from organisational data domains. All four participants receive the same controlled case description, which is presented in Appendix E.2.

The assigned task covers the workflow from the initial request up to the transition to implementation. Participants use the tool to clarify the request, generate and review the proposal, specifications, design, and implementation tasks, and examine the validation results. They are not required to implement or deploy a complete data product within the session.

Each participant works in a separate copy of the same Declarative Automation Bundle (DAB) template. Before the session, the researcher prepares the complete Visual Studio Code environment. The DAB project is open, the proposed tool is installed, and GitHub Copilot is available. Participants therefore do not need to install or configure the development environment themselves.

The evaluation uses the version of the proposed tool produced after the technical demonstration. This version extends OpenSpec with the `clarification`, `extend-propose`, `extend-specs`, and `validation` skills, together with the predefined Databricks skill. Before the evaluation, the `configure-openspec` skill is used to create an organisation-specific `development-requirements` skill. This skill is based on the technical and governance standards for data-product development described in the case organisation's documentation. The `configure-openspec` skill then integrates the extension skills, Databricks skill, and development-requirements skill into each copy of the DAB project.

5.2.2 Session procedure

At the start of each session, the participant receives the instructions presented in Appendix E.2. The researcher asks the participant to use the proposed tool to complete the assigned task and explains that the participant should verbalise expectations, uncertainties, and reactions throughout the session.

The participant then starts the workflow from the provided initial request. The tool presents clarification questions, describes the workflow stages, and guides the participant through artefact generation and

review. The participant answers the questions and reviews the generated proposal, specifications, design, and implementation tasks. The researcher observes the interaction and records participant comments and workflow observations in session notes. The researcher does not comment on the generated outputs or guide the participant's decisions. The only intervention consists of reminding the participant to continue verbalising when they become silent.

The participants are not required to continue to implementation. P3 chooses to initiate the implementation phase after completing the specification workflow. The available session time is insufficient to complete the implementation. Following the controlled task, each participant completes the adapted Technology Acceptance Model questionnaire described in Chapter 3.

Figure 5.2 illustrates the workflow during the controlled evaluation. After the participant provides the initial request, the programming agent identifies the relevant Databricks and organisation-specific skills and presents targeted clarification questions before generating the proposal. After the proposal is created, the workflow continues to a post-artefact review checkpoint before specification generation.

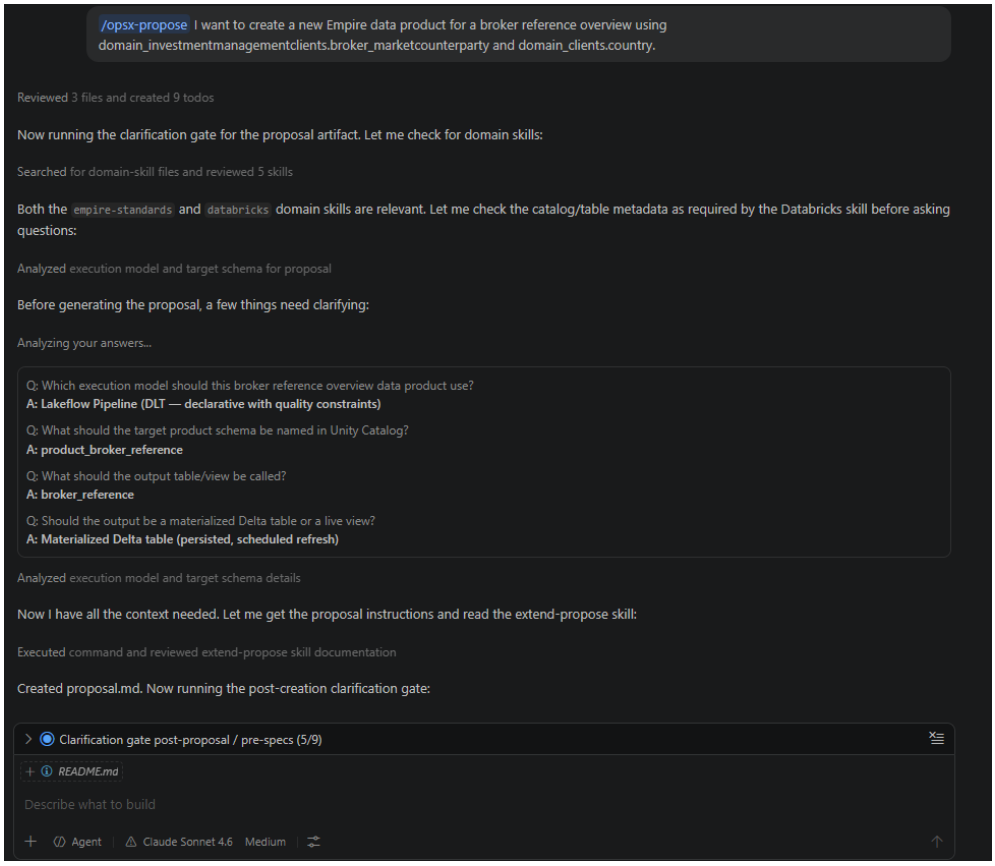


Figure 5.2: Example of the proposed tool during the controlled evaluation, showing domain-skill selection, clarification questions, proposal generation, and a post-artefact review checkpoint.

5.2.3 Think-aloud observations

The think-aloud observations concern how participants move through the workflow, respond to the generated artefacts, and identify missing or unclear information. Table 5.3 summarises the main observations and indicates the sessions from which the evidence originates.

Development workflow

The participants use different development approaches before the evaluation. P1 and P2 generally follow a development-first approach, P3 follows a mixed approach, and P4 follows a specification-first approach. All four participants progress from the initial request through clarification, artefact generation, validation, and review. P3 also initiates the implementation phase, although the implementation is not

Theme	Observation	Evidence
Development workflow	Participants with both development-first and specification-first approaches showed appreciation for the workflow	P1, P2, P3, P4
Documentation	Participants appreciated the comprehensive documentation and consistent project structure generated before implementation.	P1, P2, P3, P4
Perceived usefulness	Participants expected the tool to reduce development effort and improve documentation quality.	P1, P2, P3, P4
Learning curve	Initial familiarisation with GitHub Copilot and the generated artefacts was required.	P1, P3, P4
Limitations	Missing clarification of source information, join conditions, transformation logic and the data model was observed.	Researcher, P3, P4

Table 5.3: Summary of findings from the think-aloud evaluation sessions

completed within the available time. All participants respond positively to generating specifications before implementation. P1 and P2 indicate that this provides a more structured starting point and reduces the amount of documentation that normally needs to be created afterwards. P4 considers the generated artefacts sufficiently detailed for an existing specification-first approach.

Documentation

All participants respond positively to the generated documentation and consider the artefacts structured and detailed. They value that the workflow generates documentation before implementation and uses a consistent structure across the proposal, specifications, design, and implementation tasks. P1 specifically highlights that the documentation records design trade-offs and remarks that the workflow could ensure that "everyone follows the same structure." P1 also notes that documentation is normally created after implementation. P1 and P2 further observe that the artefacts address topics and considerations that they do not normally document explicitly.

Perceived usefulness

All participants indicate that the proposed workflow could support their data-product development work. They particularly expect the generated artefacts to reduce the effort required for development and subsequent documentation. P2 estimates that the workflow completes approximately "80% of the work" within a few hours. P2 also estimates that, when the data-product concept is already known, completing the product could take another two or three days.

Learning curve

Some participants initially require time to become familiar with the workflow and generated artefacts. P1 is at times uncertain whether an activity is performed by the programming agent or requires a manual action. P1 also comments that the amount of generated text requires familiarisation. Initial familiarisation with the workflow and artefact structure is also observed for P3 and P4.

Limitations

The sessions identify several cases in which information is not explicitly elicited before artefact generation. These concern source systems, source attributes, output attributes, join conditions, and transformation logic. Although the specification extensions and domain skills provide locations and instructions for documenting transformation requirements, the clarification skill does not explicitly request transformation logic when this information is missing.

A technical issue occurs when a DAB project is not connected to a Databricks workspace. The workflow attempts to use the Databricks CLI but cannot retrieve the required workspace information. It nevertheless continues to subsequent activities, allowing assumptions to be introduced in place of information from the environment.

The organisation-specific development requirements require an Entity Relationship Diagram as part of the specification. The ERD is generated during the first session but is absent from the other three sessions. The instruction is therefore followed in one of the four controlled-evaluation sessions.

5.2.4 Technology Acceptance Model

Following the controlled task, participants complete a Technology Acceptance Model (TAM) questionnaire concerning perceived usefulness, perceived ease of use, and willingness to use the proposed tool. Table 5.4 presents the average score for each statement.

Statement	Average
Perceived usefulness	
Using this tool would enable me to complete data product development tasks more quickly.	4.50
Using this tool would improve the quality of my data product development output.	4.25
Using this tool would help me complete more data product development work within the same amount of time.	4.50
Using this tool would help me better achieve the objectives of data product development.	4.25
Using this tool would make data product development easier to perform.	5.00
Overall, I would find this tool useful in data product development work.	4.50
Perceived ease of use	
Learning to operate the tool would be easy for me.	4.75
I would find it easy to get the tool to do what I want it to do.	4.50
My interaction with the tool would be clear and understandable.	4.50
I would find it easy to interact with this tool in the way I need.	4.25
It would be easy for me to become skillful at using this tool.	4.50
Overall, I would find this tool easy to use.	4.75
Willingness to use	
If this tool were available in practice, I would be willing to use it in my work.	4.25

Table 5.4: Average scores for the Technology Acceptance Model questionnaire ($n = 4$)

The average scores for the six perceived-usefulness statements range from 4.25 to 5.00. The highest average is obtained for the statement that the tool would make data-product development easier to perform, which receives a score of 5.00.

The six perceived-ease-of-use statements receive average scores between 4.25 and 4.75. The statements concerning learning to operate the tool and its overall ease of use both receive an average score of 4.75. The statement concerning willingness to use the tool receives an average score of 4.25.

5.2.5 Refinements

The controlled evaluation identified several opportunities for further improving the proposed tool. In contrast to the refinements resulting from the technical demonstration, the observations made during the organisational evaluation primarily concerned the completeness of requirements elicitation and the presentation of generated artefacts. None of the observations questioned the overall architecture of the proposed workflow. Instead, they resulted in refinements to the clarification skill and the domain skills – Databricks skill and development requirement.

Table 5.5 shows that the refinements affect three existing components of the proposed tool. The clarification skill is extended to request missing source information, output attributes, join conditions, and transformation logic. The Databricks skill is extended to verify the DAB workspace connection before continuing with Databricks CLI activities, while the organisation-specific development-requirements skill is updated to require the Entity Relationship Diagram explicitly. The resulting version of the proposed tool is used in the practical evaluation.

5.3 Practical Evaluation

Following the controlled evaluation and the resulting refinements, the proposed tool is applied to ongoing development activities within the case organisation. Whereas the controlled evaluation uses the same

Observation	Implemented refinement
Data sources and their attributes were assumed when not explicitly provided.	The clarification skill was extended to explicitly request the data sources. When source or output attributes are not provided, the available source attributes are presented.
Output attributes were assumed when not explicitly provided.	The clarification skill was extended to explicitly request and suggest output attributes.
Join conditions were assumed during specification generation.	The clarification skill was extended to explicitly request and confirm join conditions before generating the specifications.
Transformation logic was assumed when not explicitly provided.	The clarification skill was extended to explicitly request and confirm the required transformation logic before generating the specifications.
Databricks CLI commands continued when the Declarative Automation Bundle was not connected to a workspace.	The Databricks skill was extended to verify the Declarative Automation Bundle workspace connection before executing Databricks CLI commands. If no connection is available, the programming agent is instructed to configure the workspace before continuing.
The required Entity Relationship Diagram was not consistently generated.	The organisation-specific development requirements skill was updated to explicitly require generation of an Entity Relationship Diagram during specification generation.

Table 5.5: Refinements implemented following the controlled evaluation

case and technical setup for each participant, the practical evaluation examines the workflow within the differing contexts of two ongoing development tasks.

5.3.1 Evaluation setup

The practical evaluation consists of one joint session with participants P2 and P3 and lasts two hours. Each participant works within their own existing DAB repository in Visual Studio Code and uses an initial request based on their ongoing development task. The participants therefore use different case descriptions, source information, and project contexts.

At the start of the session, OpenSpec and the proposed extension are installed in both DAB repositories. The `configure-openspec` skill is then used to integrate the extension with OpenSpec in each repository. This configuration includes the refinements resulting from the controlled evaluation and the organisation-specific `development-requirements` skill. GitHub Copilot is available as the programming agent in both Visual Studio Code environments. The installation and configuration activities form part of the two-hour session. After configuration, each participant uses the tool within their own project. The researcher remains present to observe the workflow and record participant comments and feedback. The participants operate the tool themselves, while the workflow provides instructions through its questions and review checkpoints. The session is documented through researcher notes.

The practical evaluation focuses on whether the refined workflow can be applied to the two ongoing development tasks. It does not compare the tasks directly or require either participant to complete the full implementation. At the end of the session, P2 and P3 complete the same TAM questionnaire administered after the controlled evaluation.

5.3.2 Practical evaluation observations

Workflow application

Both participants follow the workflow from the initial request through clarification, artefact generation, and review. Neither participant encounters an issue that prevents progression through the workflow. Both respond positively to the additional clarification questions introduced after the controlled evaluation and indicate that the questions address relevant aspects of their respective tasks. The clarification issues identified during the controlled evaluation concerning sources, attributes, join conditions, transformation logic, and data models are not observed again during the practical evaluation. The participants progress to different points in the workflow. P3 completes the specification workflow and initiates the

implementation phase. The implementation is not completed within the available session time. P2 generates and reviews the proposal, specifications, and design in detail and uses the review checkpoints to request changes to the generated documentation. P2 does not proceed to implementation during the session.

Contextual relevance

P3 applies the workflow to an exceptional Databricks use case that has already been under development for an extended period. P3 initially questions whether GitHub Copilot will understand the required Databricks context. The generated clarification questions and artefacts are nevertheless considered relevant to the task, and P3 indicates that the workflow addresses the Databricks context better than expected. P2 applies the workflow to a new development task and initially provides a high-level description. The workflow responds with questions concerning the information required to specify the task. P2 describes the workflow as "knowing the context" and remarks that "it already starts asking the right questions."

Practical value

Both participants indicate that they intend to continue using the generated artefacts and proposed workflow for their ongoing tasks after the session. P2 states that the clarification questions help to understand the task and plan the implementation before development begins. P3 responds positively to the relevance of the questions and artefacts for the exceptional Databricks use case. Both participants indicate that they would use the proposed workflow alongside complementary Databricks tooling, such as Genie, for specific development activities. No additional technical or workflow issue requiring a refinement is identified during the session. One remaining usability suggestion concerns allowing users to request additional explanations of the presented clarification options before selecting an answer.

5.3.3 Follow-up Technology Acceptance Model

Following the practical evaluation, P2 and P3 complete the same thirteen-item TAM questionnaire used in the controlled evaluation. Table 5.6 compares their scores after both evaluation stages. Perceived usefulness and perceived ease of use are calculated as the average of the six statements in each category. Willingness to use is measured through one statement.

Participant	Evaluation stage	Perceived usefulness	Perceived ease of use	Willingness to use
P2	Controlled evaluation	4.17	3.83	4.00
P2	Practical evaluation	4.17	3.83	4.00
P3	Controlled evaluation	4.17	4.83	4.00
P3	Practical evaluation	4.50	4.00	4.00

Table 5.6: TAM construct scores for P2 and P3 after the controlled and practical evaluations

P2's scores remain unchanged between the controlled and practical evaluations. P3's perceived-usefulness score increases from 4.17 to 4.50, while the perceived-ease-of-use score decreases from 4.83 to 4.00. The willingness-to-use score remains 4.00 for both participants across both evaluation stages. Because the follow-up includes only two participants, these scores are reported descriptively.

6 Discussion

This chapter interprets the design and evaluation findings in relation to the research objective of developing a Spec-Driven Development tool that supports efficient data-product development. The discussion first considers what the successive evaluation stages establish about the proposed artefact. It then relates the findings to Spec-Driven Development and vibe coding, including the role of developer involvement and the non-deterministic nature of AI-generated outputs. Subsequently, it discusses the applicability of the artefact to data-product development and its practical and design implications. Finally, the limitations of the study are considered.

6.1 Overall interpretation of the evaluation findings

The three evaluation stages provide progressively broader but differently bounded evidence about the proposed artefact. The technical demonstration established that its components could be integrated into an executable workflow, while also showing that the availability of individual components was insufficient without explicit orchestration. The controlled evaluation showed that practitioners perceived the resulting process and artefacts as useful, but revealed that apparently comprehensive specifications could still contain unsupported assumptions when essential information had not been elicited. The practical evaluation subsequently indicated that the refined workflow could be applied to two ongoing development tasks.

Taken together, the findings suggest that the principal contribution of the artefact lies in introducing procedural structure around an LLM-based programming agent. Clarification, contextual grounding, explicit sequencing, validation, and review points reduce the number of development decisions left implicitly to the agent. They do not, however, guarantee that generated requirements, design decisions, or implementations are complete or correct. Human review and complementary technical validation therefore remain necessary.

The evaluation consequently supports the technical feasibility, perceived usefulness, and preliminary practical applicability of the artefact to the specification and initial development of data products. Because implementation was not completed and no comparative baseline was used, objective improvements in development efficiency, implementation quality, or data-product quality remain to be established.

6.2 Implications for Spec-Driven Development

This section discusses what the findings indicate about Spec-Driven Development in AI-assisted development. It first examines how specifications guide design, task generation, implementation, and validation. It then considers how the proposed workflow retains potential benefits of vibe coding while addressing some of its limitations. Finally, it discusses why structured clarification and review do not replace developer judgement, particularly because the outputs of programming agents remain non-deterministic.

6.2.1 Specifications as driver for implementation

The proposed artefact reflects the main elements of the working definition of SDD developed in Chapter 2. Specifications function as primary development artefacts, are created before implementation, and describe what the data product should do rather than how it should be implemented. Technical implementation decisions are documented separately in the design artefact. The specifications subsequently provide the basis from which the programming agent derives the design, implementation tasks, and code.

The evaluation provides stronger evidence for some of these elements than for others. Participants follow the sequence of clarification, artefact generation, and review, and P3 initiates implementation using the generated artefacts. This shows that the clarification process can shape the specifications and that the resulting artefacts can guide planning and initial implementation. However, because implementation is

not completed during the practical evaluation, the study does not establish whether the specifications consistently guide the generation and verification of a complete data product.

The comparison with IEEE 830 showed that SDD specifications largely build on established Software Requirements Specification content. Of the 18 specification elements identified in the SDD literature, seven had a direct counterpart in IEEE 830 and eight had a partial counterpart. This substantial overlap suggests that SDD does not introduce an entirely new type of requirements specification. Instead, it changes the role of specifications within the development process. In addition to documenting and communicating requirements, specifications provide the context from which the programming agent derives subsequent development artefacts and implementation activities.

The findings therefore suggest that the distinctive contribution of SDD lies less in introducing new requirements content and more in giving specifications a central and operational role in AI-assisted development. In the proposed workflow, the specification is not only a description of the intended system but also an input that actively guides design, task generation, implementation, and validation.

6.2.2 SDD as a response to vibe coding

The reviewed literature presents SDD as a more structured response to the benefits and limitations of vibe coding. Both approaches use natural-language interaction and generative AI to support software development. However, vibe coding is characterised by iterative prompting and limited review of generated code, whereas SDD introduces persistent and reviewable specifications that guide implementation and verification [16, 20, 23, 39]. The proposed artefact retains the natural-language interaction associated with vibe coding but changes how the initial request is processed. Instead of moving directly from a high-level prompt to generated code, the request is clarified and progressively transformed into proposal, specification, design, and task artefacts. These artefacts provide persistent representations of the requirements and development decisions and are reviewed before they are used for implementation.

Retaining the potential benefits of vibe coding

The evaluation provides some support for the potential *rapid prototyping* and *productivity* benefits identified in Table 2.2. One participant, P2, estimates that approximately "80% of the work" is completed within several hours and expects that completing the data product could take another two or three days when its concept is already known. These estimates, together with the feedback from the other participants, indicate that participants expect the generated artefacts and implementation support to reduce manual effort and accelerate the progression from an initial idea towards implementation. However, these are participant expectations rather than objectively measured productivity improvements. Because no implementation is completed during the practical evaluation, the study does not demonstrate that the artefact enables the rapid production of functional data products. The workflow may also support *accessibility*. Instead of requiring users to formulate a single complete and technically detailed prompt, the clarification process divides requirements elicitation into smaller and more focused questions. Participants with limited GitHub Copilot experience were able to follow the workflow, although some required time to become familiar with it. Because most participants already had substantial Databricks experience, the findings indicate accessibility for practitioners with domain knowledge but limited experience with AI-assisted development rather than for users with limited programming or data-engineering expertise generally. The generated artefacts also included constraints, trade-offs, non-functional requirements, and data-model considerations that participants indicated they would not normally document or consider explicitly. This relates to the potential benefit of *experimentation and creativity*, as the programming agent can introduce additional considerations and alternatives before implementation. However, the evaluation did not assess whether participants generated or evaluated more alternatives than during their normal development process. Experimentation and creativity should therefore be regarded as potential rather than demonstrated benefits. The consistent artefact structure may similarly support *collaboration* by giving developers a shared basis for discussing requirements and design decisions. This may be particularly relevant when developers normally use different approaches to specification and documentation. Nevertheless, the evaluation did not observe collaborative development or the use of the artefacts by people with substantially different technical backgrounds. Improved collaboration therefore cannot be concluded from the current findings.

Addressing process-related limitations

The proposed artefact also introduces mechanisms intended to address several process-related limitations of vibe coding. Persistent proposal, specification, design, and task artefacts make requirements and decisions more visible and may thereby address *limited accountability*. Review points require users to inspect and approve generated information before the workflow continues, which may reduce *over-reliance on AI*. The inclusion of alternatives and trade-offs creates opportunities to reconsider generated decisions and may reduce *premature convergence*. In addition, validation is intended to identify unsupported assumptions and examine the relationship between the provided context, specifications, and tasks, thereby addressing *limited quality assurance* at the artefact level.

These mechanisms do not eliminate the underlying limitations. Predefined clarification options may still encourage *premature convergence* when users select a suggested answer without considering alternatives that were not presented. Review points may reduce *over-reliance on AI*, but they do not ensure that users critically examine or understand the generated artefacts. Persistent documentation may improve *accountability*, but the completeness of decision traceability was not systematically measured. Similarly, the validation performed in this study examined the relationship between context, specifications, and tasks. It does not replace code review, testing, or independent technical validation and therefore only partially addresses *limited quality assurance*.

Remaining technical limitations

The workflow also does not eliminate the technical limitations associated with vibe coding. Explicit requirements and constraints may reduce the likelihood of *uneven code quality*, but the evaluation did not systematically assess whether the generated code was correct or consistent. Security requirements can be included in the specifications, but the resulting implementation was not evaluated for *security risks*. Persistent documentation may help address *maintainability problems* by preserving intended behaviour and relevant design decisions, but maintainability was not evaluated over time. The specifications and design artefacts may also provide context during error resolution, but the evaluation did not include an extended debugging process and therefore provides no evidence that the artefact reduces *debugging difficulties*.

Overall, the proposed artefact introduces mechanisms intended to preserve the accessibility and development support associated with natural-language interaction while addressing several process-related limitations of vibe coding. The evaluation supports the operation and perceived usefulness of these mechanisms, but does not establish that they improve code quality, security, maintainability, collaboration, or development efficiency in practice.

6.2.3 The continuing role of the developer

The vibe-coding literature indicates that experience affects how users formulate prompts for programming agents. Geng et al. [24] found that advanced students provided more information about intended functionality and the existing codebase than introductory students. The structured clarification process partly externalises this prompting activity by asking users to provide or confirm missing context through focused questions. It may therefore reduce the extent to which users must independently formulate a complete and context-rich prompt.

However, structured questions and review points do not remove the need for technical and domain expertise. Users must still determine whether generated requirements and proposed technical choices are appropriate. They may also accept professionally structured outputs without fully understanding their consequences. Although the workflow increases developer involvement in specification and design decisions, the evaluation did not establish whether participants inspected or understood the generated code more closely.

Developer involvement remains particularly important because LLM-based programming agents are non-deterministic. Identical initial descriptions may produce different questions, specifications, tasks, or implementations depending on the conversation history, available project context, and underlying model or tool. The artefact cannot eliminate this variability. Instead, it introduces procedural consistency by defining a sequence of clarification, artefact generation, review, validation, and implementation and by standardising the categories of information to be documented.

The evaluation also showed that these controls cannot guarantee that instructions will always be followed. For example, the required Entity Relationship Diagram was not consistently generated, and the initial workflow continued with Databricks CLI activities when no valid workspace connection was available. The contribution of the artefact should therefore be understood as increased process standardisation rather than deterministic output generation. Developers remain responsible for verifying the generated requirements, technical decisions, and implementation outcomes.

6.3 Applying SDD to data-product development

This section discusses what is required to apply SDD to analytical or integrated data-product development. It first examines the specification of data sources, attributes, transformations, relationships, and data models. It then discusses the need to integrate platform-specific and organisational knowledge into the workflow. Finally, it defines the boundaries of the demonstrated applicability in relation to data-product types, characteristics, and lifecycle stages.

6.3.1 Required specification content

This study focused specifically on analytical or integrated data products. As described in Chapter 2, this type of data product combines or exposes data across domains to support cross-domain analysis. Specifying such a product requires information about the relevant data sources, the data to be selected, how data from different sources should be combined, the transformations to be performed, and the structure of the resulting product. The controlled evaluation confirmed the importance of this information. Although participants initially perceived the generated artefacts as comprehensive, the evaluation identified missing clarification concerning source systems, source and output attributes, join conditions, transformation logic, and the data model. These gaps concerned essential requirements because they determine which data enters the product, how the data is combined and transformed, and what information is ultimately made available.

The additional clarification questions introduced after the controlled evaluation directly addressed these observed information gaps. Rather than allowing the programming agent to select sources, attributes, join conditions, transformations, or output structures based on its own interpretation, the refined workflow asks the user to provide or confirm this information. This makes the specifications more closely grounded in the intended data product. However, identical cases were not evaluated before and after the refinements. It therefore cannot be concluded that the additional questions ensure complete specifications for every data product.

The practical evaluation provides preliminary evidence that the resulting clarification and specification structure can be used in ongoing development work. P2 used the questions and generated artefacts to structure and plan a new task, while P3 applied the workflow to an exceptional Databricks use case. Both participants considered the questions and generated artefacts relevant to their development contexts. This indicates that the structure was not limited to the representative case used during the development of the artefact.

Using IEEE 830 to identify gaps in OpenSpec

The earlier mapping between technical data-product components and IEEE 830 was used to identify which requirements categories should be represented in the proposed artefact. The missing information identified during the controlled evaluation provides practical support for this mapping. Source systems and attributes relate to external interface and data requirements, join conditions and transformation logic relate to functional requirements, and the data model relates to logical database requirements. These findings indicate that functional behaviour alone is insufficient for specifying analytical or integrated data products.

OpenSpec provided a basis for documenting the purpose, scope, functional behaviour, and some input and output behaviour of a proposed change. However, it did not structurally require several categories identified through the mapping, including contextual information, logical database requirements, performance requirements, design constraints, reliability, availability, and security. The proposal and specification artefacts were therefore extended to provide explicit locations for these requirements.

The evaluation provides the strongest support for the extensions concerning data inputs, outputs, transformations, relationships, and logical data structures. The relevance of the remaining categories is primarily supported by the literature-based mapping, as their completeness and practical value were not systematically evaluated.

6.3.2 The need for contextual grounding

The findings show that applying SDD to analytical or integrated data-product development requires contextual knowledge beyond generic specification templates. During the tool comparison, the evaluated SDD tools made assumptions about source attributes, architectures, infrastructure, permissions, and cloud components that did not correspond with the case environment. These assumptions arose because the programming agents attempted to complete the specifications without sufficient knowledge of the platform or organisation. The proposed artefact addresses this problem through separate sources of contextual knowledge. The Databricks skill provides platform-specific guidance and, where available, information from the actual development environment. The organisation-specific development-requirements skill contains standards and constraints from the case organisation. The Databricks skill therefore describes how development can be performed using the available technology, while the development-requirements skill specifies how data products should be developed within the organisation.

The development process also showed that making contextual knowledge available does not ensure that it will be used. The initial Databricks skill was present in the development environment but was not automatically invoked by OpenSpec. Consequently, the programming agent could proceed without considering the available platform knowledge. This led to the introduction of the configure-openspec component, which connects the contextual skills to the workflow. Contextual grounding therefore requires both the availability of relevant knowledge and a mechanism for identifying when that knowledge is applicable and invoking it at the appropriate stage. Separating the generic workflow, platform knowledge, and organisational requirements also creates a potentially transferable design. In principle, the same core workflow could be adapted to another platform or organisation by replacing or extending the contextual skills without redesigning the complete process. However, only the Databricks implementation and the requirements of the case organisation were evaluated. Transferability to other technical and organisational contexts therefore remains a design possibility rather than a demonstrated finding.

6.3.3 Scope of data-product applicability

The definition presented in Chapter 2 describes a data product as a managed artefact that satisfies recurring information needs and creates value by transforming and packaging relevant data into a consumable form [26]. Data products are further characterised by a recurring information need, a well-defined consumer base, measurable value, the combination of data from multiple sources, and delivery in a consumable form. The proposed artefact primarily supports the technical activities required to combine, transform, and package data. This is particularly relevant to analytical or integrated data products because they combine or expose data across domains. However, the evaluation did not systematically assess whether the specified products addressed recurring information needs, had a well-defined consumer base, or created measurable value. Because the products were not followed through delivery and consumption, the study also does not establish whether the resulting outputs ultimately met consumer requirements. The findings therefore provide stronger evidence for the technical construction of data products than for their value-oriented and consumer-oriented characteristics.

The background identifies several additional qualities that data products should exhibit: they should be discoverable, addressable, understandable, accessible, interoperable, reusable, secure, and valuable. Specification categories included in the artefact may support these qualities. For example, metadata and semantic definitions can contribute to discoverability and understandability, interface requirements can support addressability and interoperability, and access and security requirements can support accessibility and security. Nevertheless, the evaluation did not measure whether the resulting data products exhibited these qualities. Documenting a requirement for a quality should therefore not be interpreted as evidence that the quality was achieved.

The demonstrated applicability is also bounded by the type of data product examined. The data-product literature distinguishes between basic or fundamental, analytical or integrated, and advanced analytical data products, as presented in Table 2.10. This study focused exclusively on analytical or integrated products. Basic or fundamental products may require a lighter specification process because

their logic and intended use may be less complex. Advanced analytical products may require additional requirements concerning model behaviour, training data, evaluation measures, explainability, bias, and analytical performance. Since neither type was included in the evaluation, the adaptations required for them cannot be determined from the current findings.

Finally, the evaluation covered only part of the data-product lifecycle presented in Figure 2.3. The proposed artefact primarily supports initiation and build by clarifying the intended change and generating specifications, design decisions, and implementation tasks. Publication, access requests, delivery, consumption, refinement, and decommissioning were outside the evaluated workflow. The study consequently provides no direct evidence regarding publication in a data catalogue, consumer access through the data shop, continued SLA/SLO management, incident handling, or refinement based on consumer needs. The demonstrated applicability should therefore be limited to the technical specification and initial development of analytical or integrated data products within the examined Databricks context. The findings do not establish support for the complete data-product concept, the other two product types, or the full data-product lifecycle.

6.4 Practical and design implications

This section considers what the findings mean for the practical use and further design of SDD workflows. It first discusses the benefits and maintenance requirements of the generated documentation and the extent to which it supports traceability. It then considers the learning curve, initial acceptance, and use of the artefact alongside existing tools. Finally, it presents design implications concerning clarification, workflow orchestration, contextual knowledge, validation, specification detail, and human responsibility.

6.4.1 Documentation and traceability

Documentation was one of the clearest positive themes in the controlled evaluation. Participants valued the comprehensiveness and consistent structure of the generated artefacts, the explicit documentation of trade-offs, and the production of documentation before rather than after implementation. One participant particularly valued that the workflow could ensure that "everyone follows the same structure."

The proposal, specifications, design, and implementation tasks document the intended change before implementation begins. This makes the requirements and technical decisions available for review at an earlier stage and may reduce the need to reconstruct them after implementation. However, the controlled evaluation showed that detailed artefacts are not necessarily complete. Participants initially perceived the generated artefacts as comprehensive, even though essential information was still missing. The evaluation also did not systematically examine whether the information in the proposal, specifications, and design could be traced through the implementation tasks to the resulting code. The artefacts therefore provide a basis for traceability, but the study does not demonstrate that this traceability was complete.

The generated artefacts also require effort to review and maintain. Several participants initially considered the amount of text substantial. If the implementation changes without corresponding updates to the specifications and design, these artefacts may become outdated and reduce rather than improve clarity. Their long-term alignment with the implementation was not evaluated.

6.4.2 Learning curve, adoption, and complementary tools

Although participants generally perceived the workflow as easy to use, the think-aloud observations showed that initial familiarisation was required. Participants with limited GitHub Copilot experience sometimes needed guidance and were uncertain about which activities would be performed by the programming agent and which actions remained their own responsibility. The number and structure of the generated artefacts were also unfamiliar to some participants. These observations may partly result from limited familiarity with GitHub Copilot. However, the proposed artefact also introduces its own workflow steps, artefacts, clarification questions, and review points. The current evaluation cannot fully separate the learning required for GitHub Copilot from the learning required for the workflow itself. Adoption may therefore require onboarding that explains the workflow sequence, the role of each artefact, when the agent will continue, and which decisions require human approval.

Despite this initial learning requirement, participants express a positive willingness to use the artefact. The average willingness-to-use score after the controlled evaluation is 4.25, and both participants in the

practical evaluation indicate that they intend to continue using the workflow. However, the follow-up TAM results do not show uniform improvements. P2's scores remain unchanged, while P3's perceived-usefulness score increases from 4.17 to 4.50 and the perceived-ease-of-use score decreases from 4.83 to 4.00. Willingness to use remains unchanged for both participants. The practical application does not produce a uniform improvement in acceptance scores. Because the follow-up includes only two participants, the results are too limited to explain these changes. Furthermore, willingness expressed immediately after an evaluation does not necessarily result in sustained use. Continued adoption depends on whether the benefits experienced during regular development compensate for the effort required to clarify requirements, review artefacts, and keep them aligned with implementation.

The artefact is also likely to be used alongside rather than instead of existing development tools. Participants indicated that they would continue using specialised Databricks AI tools such as Genie for data exploration and platform-specific analysis. The artefact is therefore best positioned as a specification and workflow layer within the existing development environment rather than as a replacement for data-development tools.

6.4.3 Design principles for SDD workflows

Several broader design implications can be derived from the findings. First, clarification should be treated as a central and iterative part of SDD. The completeness of specifications depends on the information available to the programming agent. When essential information is missing, the workflow should expose this uncertainty and request user input rather than silently converting assumptions into requirements or implementation decisions. Users should also be able to refine requirements after reviewing generated proposals, specifications, or designs.

Second, the stages of an SDD workflow must be explicitly orchestrated. Contextual extensions should be applied before validation, and validation should occur before an artefact is used for subsequent development activities. Review points should also prevent the programming agent from continuing before the user has inspected the relevant output. Explicit sequencing reduces the likelihood that unsupported interpretations propagate from the initial request into specifications, tasks, and implementation.

Third, contextual knowledge must be actively and modularly integrated. Making platform or organisational knowledge available is insufficient if the programming agent does not recognise when it should be applied. The workflow should therefore connect relevant contextual knowledge to specific stages and verify that it has been considered. Separating generic workflow logic, platform knowledge, and organisational requirements can support adaptation while preventing context-specific rules from becoming embedded in the core workflow.

Fourth, validation should distinguish between traceability and correctness. A requirement can be traced to user input while still being technically incorrect, while a technically plausible requirement may remain unsupported by the available information. AI-based validation can identify inconsistencies and unsupported assumptions but does not provide an independent guarantee of correctness. Schema validation, environmental checks, automated testing, code review, and human verification remain necessary.

Fifth, the level of specification detail should be proportionate to the development task. Comprehensive documentation can support review and traceability but also increases the amount of material that developers must inspect and maintain. Optional or context-dependent specification sections may provide a better balance between comprehensiveness and usability than requiring every category for every task.

Finally, human responsibility should remain explicit throughout the workflow. SDD shifts part of the developer's effort from manually producing implementation details towards specifying, reviewing, and approving the context used by the programming agent. A structured workflow can support this responsibility but cannot guarantee critical engagement. Developers remain responsible for verifying requirements, technical decisions, and implementation outcomes.

6.5 Limitations

Several limitations affect what can be concluded from this study. The evaluation provides evidence that the proposed tool is technically feasible, is perceived as useful, and can support the initial development of analytical or integrated data products. However, it does not establish that the tool consistently improves development efficiency or implementation quality.

First, the evaluation includes a small group of participants from one department within the same organisation. The controlled evaluation consists of four participants working with one case during sessions of approximately one hour. The practical evaluation includes two of these participants and two ongoing development tasks during a single two-hour session. This provides relevant practitioner feedback but limits the extent to which the findings can be applied to other teams and organisations. Neither evaluation follows the participants through the complete implementation and deployment of a data product. The study therefore does not show whether the tool remains useful during longer-term use or whether the generated artefacts remain aligned with the implementation over time.

Second, the study does not include a baseline task without the proposed tool or objective measurements of development time, rework, or implementation quality. The indications of improved efficiency mainly come from participant expectations, think-aloud comments, and TAM responses. These results support perceived usefulness but do not prove that development becomes more efficient. The refinements made after the controlled evaluation are also applied to different practical tasks rather than evaluated through a direct before-and-after comparison. Their individual effects can therefore not be determined.

Third, the qualitative analysis depends partly on researcher interpretation. Researcher notes form the main record of the controlled and practical evaluations. Audio recordings provide additional support where available, while the practical evaluation relies only on notes. The same researcher develops the artefact, conducts the evaluations, and groups the observations into themes. Although the researcher does not guide participant decisions, this involvement may influence which observations receive attention and how they are interpreted.

Fourth, the findings depend on the examined technical environment. The programming agent is non-deterministic, meaning that repeated use with the same input may produce different questions, artefacts, and implementation decisions. The observed results therefore describe the evaluated executions rather than guaranteed tool behaviour. The implementation also depends on the examined versions and configuration of GitHub Copilot, Claude Sonnet 4.6, OpenSpec, and Databricks. Changes to these technologies may affect how the tool operates.

Finally, the study focuses on analytical or integrated data products, the initiation and build phases, and the context of one organisation using Databricks. The findings do not establish that the tool supports other data-product types, other technical platforms, or the complete data-product lifecycle. Although the modular design allows platform and organisational knowledge to be replaced or extended, this transferability requires further evaluation.

7 Conclusion

This chapter concludes the thesis by bringing together the findings from the design, demonstration, controlled evaluation, and practical evaluation. The findings show how Spec-Driven Development can provide more structure to AI-assisted data-product development through clarification, specifications, contextual knowledge, validation, and developer review. They also show that the proposed tool is technically feasible, is perceived as useful, and can be applied to ongoing development tasks. However, the study does not establish that the tool improves development efficiency or implementation quality.

In this chapter, we first answer the research questions, then present the main research and practical contributions, and finally describe directions for future work.

7.1 Answers to the research questions

This study examines how a Spec-Driven Development tool can support efficient data-product development. The six sub-questions formulated in Chapter 1 provide the conceptual and practical basis for answering the main research question.

SQ.1: How can Spec-Driven Development be defined?

Spec-Driven Development is a software development approach in which specifications act as the primary development artefact. They are created before implementation, describe what the system should do rather than how it should be implemented, and guide the generation and verification of the implementation by a programming agent. Specifications therefore serve not only as documentation, but also as an active input to the development process.

SQ.2: What are the characteristics of specifications in Spec-Driven Development?

SDD specifications use natural language to describe intended behaviour, functionality, inputs, outputs, constraints, edge cases, and acceptance criteria. They can also include supporting artefacts such as design documents, decision reasons, and implementation tasks. Most of the identified specification elements have a direct or partial counterpart in the IEEE 830 Software Requirements Specification structure. SDD specifications can therefore largely be understood as natural-language software requirements specifications. Their main difference lies in their active role in guiding planning, implementation, and validation.

SQ.3: What are the characteristics of data-product development?

Data-product development involves more than implementing functional behaviour. It also requires information about data, metadata, transformation logic, infrastructure, policies, data quality, service expectations, access, and incident management. For analytical or integrated data products, the selected data sources, source and output attributes, join conditions, transformations, and logical data model are especially important. Data products are developed and managed throughout a wider lifecycle, but this study examines only the initiation and build phases.

SQ.4: What design insights can be derived from existing Spec-Driven Development tools?

The examined tools all connect specifications to implementation, but differ in their workflow structure, generated artefacts, clarification support, contextual grounding, data-model support, and validation. The comparison shows that general-purpose SDD tools can introduce unsupported assumptions when requirements or technical context are missing. A tool for data-product development should therefore

ask targeted questions, use platform-specific knowledge, make the data model explicit, validate generated artefacts, and provide clear review points. OpenSpec provides a suitable foundation because its lightweight and modular structure can be extended without replacing its main workflow.

SQ.5: What requirements should a Spec-Driven Development tool satisfy for data-product development?

The tool should clarify missing information before specifications are generated and should make data sources, attributes, relationships, transformations, and output structures explicit. It should extend general specifications with relevant data-product requirements and actively apply platform-specific and organisation-specific knowledge. The workflow should also validate whether requirements are supported by the available context and whether implementation tasks are linked to the specifications.

These components must be placed in an explicit order. Contextual extensions should be applied before validation, validation should take place before an artefact is used in the next stage, and users should be able to review each artefact before the workflow continues. The design should remain modular so that additional platform or organisational knowledge can be added without changing the complete workflow.

SQ.6: How effective is the proposed tool in supporting data-product development?

The technical demonstration shows that the proposed components can be integrated into an executable OpenSpec workflow. The controlled evaluation shows that participants value the structured workflow, comprehensive documentation, consistent artefact structure, and clarification questions. The mean scores for the perceived-usefulness items range from 4.25 to 5.00, while the perceived-ease-of-use items range from 4.25 to 4.75. Willingness to use the tool receives an average score of 4.25.

The controlled evaluation also identifies missing clarification about sources, attributes, join conditions, transformation logic, and data models. These observations lead to further refinements of the clarification process. In the practical evaluation, the specific clarification issues identified during the controlled evaluation are not observed again. Both participants apply the refined workflow to ongoing development tasks without encountering an issue that prevents them from progressing. They consider the questions and generated artefacts relevant and indicate that they intend to continue using the workflow. While the small practical evaluation does not show that these issues are fully eliminated, it provides an indication that the refinements address the identified problems.

The follow-up acceptance scores do not improve uniformly. P2's scores remain unchanged, while P3's perceived usefulness increases and perceived ease of use decreases. Because only two participants complete the follow-up questionnaire, these changes cannot be explained from the available evidence. The evaluation therefore supports the technical feasibility, perceived usefulness, and preliminary practical applicability of the tool. It does not establish its effect on complete implementations, implementation quality, or long-term use.

RQ: How can a Spec-Driven Development tool be designed to support efficient data-product development?

A Spec-Driven Development tool can be designed by placing a structured, specification-first workflow around a programming agent. The proposed design combines targeted clarification, data-product-specific specification content, platform and organisational knowledge, explicit workflow sequencing, validation, and human review. The initial request is gradually translated into a proposal, specifications, design decisions, implementation tasks, and initial implementation. This reduces the number of decisions that the programming agent must make from incomplete context.

Participants expect this approach to reduce development and documentation effort and perceive the proposed tool as useful. However, the study does not include a baseline task or objective measurements of development time, rework, or implementation quality. It therefore shows how the tool is designed to support efficient development and provides indications of possible efficiency benefits, but it does not prove that data-product development becomes more efficient.

7.2 Contributions

The first research contribution is a working definition of Spec-Driven Development based on recurring elements in the reviewed literature. The study also identifies the main characteristics of SDD specifications and relates them to the IEEE 830 Software Requirements Specification structure. This shows that much of the content of SDD specifications builds on established requirements practices, while their central role in AI-assisted implementation is more distinctive.

The second contribution is the mapping between technical data-product components, IEEE 830 requirements sections, and the artefacts provided by OpenSpec. This mapping identifies which requirements are already supported and which need to be added or made more explicit for analytical or integrated data products. The evaluation provides the strongest practical support for explicitly specifying data sources, attributes, join conditions, transformation logic, and logical data structures.

The third contribution is a set of design principles for SDD workflows. Clarification should be iterative, contextual knowledge should be actively connected to the workflow, and workflow stages should be explicitly ordered. Validation should distinguish between traceability and correctness, while the amount of specification detail should match the task. The findings also show that detailed documentation is not necessarily complete and that traceable information is not necessarily correct. Human review therefore remains necessary throughout the workflow.

The practical contribution is an implemented extension of OpenSpec for AI-assisted data-product development. The extension adds clarification, proposal and specification extensions, validation, configuration, Databricks guidance, and organisation-specific development requirements. It provides a concrete example of how a general SDD workflow can be adapted to a specific technical and organisational context while preserving its modular structure.

7.3 Future work

Future research should first evaluate the tool with a larger and more varied group of practitioners. A longer-term study should follow data products from the initial request through implementation, deployment, and continued maintenance. Comparing development with and without the tool would allow future studies to measure development time, manual effort, rework, implementation quality, and the effort required to keep specifications aligned with the code. This is needed to determine whether the expected efficiency benefits occur in practice.

Further research should also examine the reliability of the workflow across repeated executions. Because programming agents are non-deterministic, the same request may produce different questions, specifications, and implementation decisions. Repeated tests can show how consistently the workflow follows its instructions and includes required artefacts. Validation can also be strengthened through source-schema checks, environment checks, automated tests, and systematic comparison between requirements, tasks, and implemented code.

Finally, the tool should be evaluated with other teams, organisations, technical platforms, and data-product types. Basic data products may require a lighter workflow, while advanced analytical products may require additional requirements for models, training data, explainability, bias, and analytical performance. Future work can also examine later lifecycle stages, such as publication, delivery, consumption, refinement, and decommissioning. This determines whether the modular design can be adapted beyond the Databricks context and the initiation and build phases examined in this study.

Overall, this thesis shows that the value of Spec-Driven Development does not come from writing specifications alone. It comes from combining specifications with clarification, contextual knowledge, explicit workflow control, validation, and active developer review. This provides a practical basis for more structured AI-assisted data-product development, while its effect on development efficiency remains to be established through further research.

Bibliography

[1] Ieee recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, 1998. doi: 10.1109/IEEESTD.1998.88286.

[2] AI Unified Process. Ai unified process marketplace. `https://github.com/AI-Unified-Process/marketplace`, 2026. Accessed: 2026-03-17.

[3] Michael Armbrust, Ali Ghodsi, Reynold Xin, Matei Zaharia, et al. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, volume 8, page 28. sn, 2021.

[4] John Atkinson-Abutridy. *Large language models: concepts, techniques and applications*. CRC Press, 2024.

[5] Azure-Samples. Spec2cloud: Spec-driven development templates to build and deploy apps/agents to azure. `https://github.com/Azure-Samples/Spec2Cloud`, 2026. GitHub repository, accessed 2026-04-23.

[6] Abubakar Bello Bada, Chizzy Ifesinachi Ede, Ibrahim Musa Mungadi, and Mubashir Haruna. Democracy in software development: The rise of vibe coding. *International Journal of Engineering and Computer Science*, 2025. doi: 10.18535/ijecs.v14i11.5292. URL `https://www.ijecs.in/index.php/ijecs/article/view/5292`.

[7] Ashish Bijawat. Data warehouse vs. data lake vs. data lakehouse vs. data mesh, June 2022. URL `https://www.linkedin.com/pulse/data-warehouse-vs-lake-lakehouse-mesh-ashish-bijawat/`. LinkedIn article.

[8] Ivo Blohm, Felix Wortmann, Christine Legner, and Felix Köbler. Data products, data mesh, and data fabric: I. blohm et al. *Business & Information Systems Engineering*, 66(5):643–652, 2024.

[9] Birgitta Böckeler. Understanding spec-driven development: Kiro, spec-kit, and tessl, October 2025. URL `https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html`. Accessed: 2026-03-04.

[10] Eric W. Bridgeford, Iain Campbell, Zijao Chen, Zhicheng Lin, Harrison Ritz, Joachim Vandekerckhove, and Russell A. Poldrack. Ten simple rules for ai-assisted coding in science. arXiv preprint arXiv:2510.22254, 2025. URL `https://arxiv.org/abs/2510.22254`.

[11] P Chen, J Yang, A Beheshti, and J Su. Towards data economy: Are products and marketplaces ready? In *Proceedings of the 48th International Conference on Very Large Data Bases (VLDB)*, pages 1–4, 2022.

[12] Dan Clarke. OpenSpec: A lightweight AI-driven spec framework. URL `https://www.danclarke.com/openspec/`.

[13] Databricks. What are declarative automation bundles?, 2026. URL `https://docs.databricks.com/aws/en/dev-tools/bundles/`. Formerly known as Databricks Asset Bundles. Last updated July 10, 2026; accessed August 12, 2026.

[14] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989. doi: 10.2307/249008.

[15] Jedd Davis, Dave Nussbaum, and Kevin Troyanos. Approach your data with a product mindset. *Harvard Business Review*, 12, 2020.

[16] Den Delimarsky. Diving into spec-driven development with github spec kit, sep 2025. URL `https://developer.microsoft.com/blog/spec-driven-development-spec-kit/`. Microsoft for Developers blog post (Sep 15, 2025).

[17] Den Delimarsky. Spec-driven development with ai: Get started with a new open source toolkit, sep 2025. URL `https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/`. GitHub Blog post (Sep 2, 2025).

[18] Mahesh Deshpande. Empowering data programs: The five essential data engineering concepts for program managers. *Journal of Engineering and Applied Sciences Technology*, pages 1–12, 06 2023. doi: 10.47363/JEAST/2023(5)235.

[19] Muhammad Shoaib Farooq, Uzma Omer, Amna Ramzan, Mansoor Ahmad Rasheed, and Zabihullah Atal. Behavior driven development: A systematic literature review. *IEEE access*, 11:88008–88024, 2023.

[20] Ahmed Fawzy, Amjed Tahir, and Kelly Blincoe. Vibe coding in practice: Motivations, challenges, and a future outlook – a grey literature review. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '26, page 212–223, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724268. doi: 10.1145/3786583.3786866. URL `https://doi.org/10.1145/3786583.3786866`.

[21] Fission AI. OpenSpec: Spec-driven development for AI coding assistants. URL `https://github.com/Fission-AI/OpenSpec`. GitHub repository.

[22] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, 2007. doi: 10.1109/FOSE.2007.14.

[23] Kiev Gama, Filipe Calegario, Victoria Jackson, Alexander Nolte, Luiz Morais, and Vinicius Cardoso Garcia. "can you feel the vibes?": An exploration of novice programmer engagement with vibe coding. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering*, ICSE-SEET '26, page 485–496, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400724237. doi: 10.1145/3786580.3786992. URL `https://doi.org/10.1145/3786580.3786992`.

[24] Francis Geng, Anshul Shah, Haolin Li, Nawab Mulla, Steve Swanson, Gerald Soosai Raj, Daniel Zingaro, and Leo Porter. Exploring student-ai interactions in vibe coding. In *Proceedings of the 28th Australasian computing education conference*, pages 45–54, 2026.

[25] GitHub. Spec Kit: Toolkit to help you get started with Spec-Driven Development, 2025. URL `https://github.com/github/spec-kit`.

[26] M Redwan Hasan and Christine Legner. Understanding data products: motivations, definition, and categories. 2023.

[27] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[28] Annika Jacobsen et al. FAIR principles: Interpretations and implementation considerations. *Data Intelligence*, 2(1–2):10–29, 2020. doi: 10.1162/dint_r_00024.

[29] Monique W. M. Jaspers, Thiemo Steen, Cor van den Bos, and Maud Geenen. The think aloud method: A guide to user interface design. *International Journal of Medical Informatics*, 73(11–12): 781–795, 2004. doi: 10.1016/j.ijmedinf.2004.08.003.

[30] Soňa Karkošková. Data mesh: Guiding principles and patterns, and data catalog architectural concept. In *2024 10th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 1485–1490. IEEE, 2024.

[31] Andrej Karpathy. There's a new kind of coding I call "vibe coding" (X post), February 2025. URL https://x.com/karpathy/status/1886192184808149383.

[32] Ralph Kimball and Joe Caserta. *The data warehouse ETL toolkit*. John Wiley & Sons, 2004.

[33] Ralph Kimball and Margy Ross. *The data warehouse toolkit: The definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[34] Kiro. Kiro. https://github.com/kirodotdev/Kiro, 2026. Accessed: 2026-03-17.

[35] Jie Li, Youyang Hou, Laura Lin, Ruihao Zhu, Hancheng Cao, and Abdallah El Ali. Vibe coding in product teams: Reconfiguring ai-assisted workflows, prototyping, and collaboration. In *Proceedings of the 5th Annual Symposium on Human-Computer Interaction for Work*, pages 1–16, 2026.

[36] Mike Loukides. *The evolution of data products*. " O'Reilly Media, Inc.", 2011.

[37] Michael R Lyu, Baishakhi Ray, Abhik Roychoudhury, Shin Hwei Tan, and Patanamon Thongtanunam. Automatic programming: Large language models and beyond. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–33, 2025.

[38] Inês Araújo Machado, Carlos Costa, and Maribel Yasmina Santos. Data mesh: concepts and principles of a paradigm shift in data architectures. *Procedia computer science*, 196:263–271, 2022.

[39] Simon Maple. Tessl launches spec-driven framework and registry to make ai coding agents reliable, September 2025. URL https://tessl.io/blog/tessl-launches-spec-driven-framework-and-registry/. Accessed: 2026-03-04.

[40] Srinivas Rao Marri. Constitutional spec-driven development: Enforcing security by construction in ai-assisted code generation. *arXiv preprint arXiv:2602.02584*, 2026.

[41] Jürg Meierhofer, Thilo Stadelmann, and Mark Cieliebak. Data products. In *Applied data science: Lessons learned for the data-driven business*, pages 47–61. Springer, 2019.

[42] Christian Meske, Tobias Hermanns, Esther Von der Weiden, Kai-Uwe Loser, and Thorsten Berger. Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda. *IEEE Access*, 13:213242–213259, 2025.

[43] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 2002.

[44] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72, 2025.

[45] NLJUG. Spec-driven development with ai: A new approach and a journey into the past, 2025. URL https://nljug.org/uncategorized/spec-driven-development-with-ai-a-new-approach-and-a-journey-into-the-past/. Accessed: 2026-03-04.

[46] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[47] Deepak Babu Piskala. Spec-driven development: From code to contract in the age of ai coding assistants. *arXiv preprint arXiv:2602.00180*, 2026.

[48] Octavian Rusu, Ionela Halcu, Oana Grigoriu, Giorgian Neculoiu, Virginia Sandulescu, Mariana Marinescu, and Viorel Marinescu. Converting unstructured and semi-structured data into knowledge. In *2013 11th RoEduNet international conference*, pages 1–4. IEEE, 2013.

[49] Richard Rutledge, Sheryl Duggins, Dan Lo, and Frank Tsui. Formal specification-driven development. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 1. The Steering Committee of The World Congress in Computer Science, 2014.

[50] Denaldi Rananda Saputra, Tranggono, and Mega Cattleya Prameswari Annisa Islami. Evaluation of a virtual reality-based introduction to hazardous and toxic waste management using the technology acceptance model. *Journal of Applied Informatics and Computing*, 9(4):1722–1729, 2025. doi: 10.30871/jaic.v9i4.10040.

[51] Jan Schneider, Christoph Gröger, Arnold Lutsch, Holger Schwarz, and Bernhard Mitschang. The lakehouse: State of the art on concepts and technologies. *SN Computer Science*, 5(5):449, 2024.

[52] Shane Sendall. Domain-driven software development-a world of transformations. In *Proceedings. 15th IEEE International Workshop on Rapid System Prototyping, 2004.*, pages 110–112. IEEE, 2004.

[53] Mohamed Rizwan Syed Sulaiman. Decentralizing data for larger organizations: A data mesh approach. *Journal of Computational Analysis & Applications*, 34(11), 2025.

[54] Tessl. Spec-driven development tile. `https://github.com/tesslio/spec-driven-development-tile`, 2026. Accessed: 2026-03-17.

[55] Thoughtworks. Spec-driven development: Unpacking one of 2025's key new ai-assisted engineering practices, December 2025. URL `https://www.thoughtworks.com/insights/blog/agile-engineering-practices/spec-driven-development-unpacking-2025-new-engineering-practices`. Accessed: 2026-03-04.

[56] Van Lanschot Kempen. Van lanschot kempen – corporate website. `https://www.vanlanschotkempen.com/nl-nl`. Accessed: 2026-02-10.

[57] Richard Y Wang. A product perspective on total data quality management. *Communications of the ACM*, 41(2):58–65, 1998.

[58] Roel Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[59] Laurie Williams, E Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, pages 34–45. IEEE, 2003.

[60] Robert Winter and Tobias Hackl. Data mesh at scale. *Exploration of current practices in large organizations. University of St. Gallen, Institute of Information Management, St. Gallen*, 2023.

[61] Tomasz Wiśniewski. Spec2cloud - spec-driven development for azure with github copilot agents, February 2026. URL `https://tomaszwisniewski.com/thoughts/azure/2026/02/17/spec2cloud-spec-driven-development-azure.html`. Accessed: 2026-04-23.

[62] Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026. doi: 10.48550/arXiv.2602.12430. URL `https://arxiv.org/abs/2602.12430`.

[63] Gehao Zhang, Zhenting Wang, and Juan Zhai. Breaking the myth: Can small models infer post-conditions too? *arXiv preprint arXiv:2507.10182*, 2025.

[64] L. Zhong et al. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024. doi: 10.1609/aaai.v38i19.30185. URL `https://ojs.aaai.org/index.php/AAAI/article/view/30185`.

[65] Chaim Zins. Conceptual approaches for defining data, information, and knowledge. *Journal of the American society for information science and technology*, 58(4):479–493, 2007.

A SDD Definitions

Table A.1: Characteristics of Spec-Driven Development identified through qualitative coding of grey literature

Characteristic	Supporting Quotes	Source
Leading: Specifications are the primary artifact of development	"The Core Principle: In spec-driven development, code is the implementation detail of the specification—not the other way around. The spec declares intent; the code realizes it."	[47]
	"A development approach where specs — not code — are the primary artifact."	[9, 39]
	"Requirements stay at the center."	[45]
	"requirements [are] the source of truth and let AI handle everything downstream."	[45]
	"In this new world, maintaining software means evolving specifications. [...] The lingua franca of development moves to a higher level, and code is the last-mile approach."	[9, 16]
First: Development begins with writing specifications	"Spec-driven development (SDD) inverts the traditional workflow by treating specifications as the source of truth and code as a generated or verified secondary artifact."	[47]
	"software designers need to write formal specifications before the code implementations."	[63]
	"Spec-driven development means writing a 'spec' before writing code with AI ('documentation first'). The spec becomes the source of truth for the human and the AI."	[9]
Specifications describe what the system should do, not how	"it specifies what must hold, not how to achieve it."	[40]
	"be intentional about the separation between functional spec and technical implementation."	[9]
	"The core of SDD goes beyond vibe coding, separating the design and implementation phases."	[55]
Specifications drive implementation	"Spec-driven development (SDD) inverts the traditional workflow by treating specifications as the source of truth and code as a generated or verified secondary artifact."	[47]
	"it's a development paradigm that uses well-crafted software requirement specifications as prompts, aided by AI coding agents, to generate executable code."	[55]
	"modern SDD embeds spec validation into continuous integration, checking every commit against the spec so drift is caught immediately rather than during quarterly reviews."	[47]
AI agents generate and verify implementations based on specifications	"AI agents can generate code from specs while self-verifying against checklists for requirements adherence."	[47]
	"requirements the source of truth and let AI handle everything downstream."	[45]
	"SDD transforms AI agents from reactive tools into proactive collaborators."	[47]
	"Spec-driven development means writing a 'spec' before writing code with AI ('documentation first'). The spec becomes the source of truth for the human and the AI."	[9]

Continued on next page

Characteristic	Supporting Quotes	Source
	"it's a development paradigm that uses well-crafted software requirement specifications as prompts, aided by AI coding agents, to generate executable code."	[55]
	"specs should be authoritative, not advisory, and that modern tooling can enforce what was previously left to human discipline."	[47]
	"SDD specs are structured so AI coding assistants can consume them."	[47]

B spec elements

Table B.1: Evidence traceability: specification elements mapped to source quotes

Specification Element	Direct Quote	Source
Descriptive specifications		
Behavior	"intended behavior"	[47]
	"behavior"	[47]
	"user-facing behavior"	[47]
	"behavior-focused"	[47]
	"describing what happens rather than how"	[47]
	"it specifies what must hold, not how to achieve it."	[40]
	"informally describe the expected behavior of software systems"	[63]
	"precise, verifiable conditions that capture program behavior"	[63]
	"behavior-oriented artifact"	[9]
	"still defines the behavior of the target software; it doesn't just describe business requirements"	[55]
Functionality	"articulates what the code should do"	[47]
	"functional specification"	[47]
	"What should the software do?"	[47]
	"Feature Specifications (spec.md) define what to build (analogous to functional requirements) while respecting constitutional constraints."	[40]
	"expresses software functionality"	[9]
	"to describe business intent rather than specific tech-bound implementations"	[55]
User Stories with Acceptance Criteria	"user stories"	[47]
	"user story with acceptance criteria"	[47]
	"acceptance criteria"	[47]
	"User stories with acceptance criteria are specifications"	[47]
	"Definition of Done is a form of spec"	[47]
	"Requirements: Structured as a list of requirements, where each requirement represents a 'User Story' (in 'As a...' format) with acceptance criteria (in 'GIVEN... WHEN... THEN...' format)"	[9]
Scenarios (BDD / Given-When-Then)	"BDD scenario"	[47]
	"scenarios"	[47]
	"Given/When/Then format"	[47]
	"traditional specs are read by humans, while SDD specs are executed as BDD scenarios, API contract tests, or model simulations—if code doesn't match, the build fails."	[47]
	"define scenarios using Given/When/Then"	[55]
Business Rules	"Business rules"	[47]
Edge Cases and Error Conditions	"edge case"	[47]
	"error conditions"	[47]

Input/Output Mappings	"input-output examples"	[47]
	"input/output mappings"	[55]
	"structured inputs and outputs"	[55]
Formal specifications		
Constraints	"constraints"	[47]
	"constraints"	[55]
	"architectural constraints"	[55]
	"human-in-the-loop governance"	[55]
	"the rationale (WHY) alongside the constraint (WHAT) and pattern (HOW)"	[40]
Invariants	"invariants"	[55]
Preconditions and Postconditions	"postcondition generation"	[63]
	"translate informal intent into postcondition assertions"	[63]
Interface Descriptions and Integration Contracts	"interface types"	[55]
	"integration contracts"	[55]
	"traditional specs are read by humans, while SDD specs are executed as BDD scenarios, API contract tests, or model simulations—if code doesn't match, the build fails."	[47]
State Machines / Sequential Logic	"sequential logic/state machines"	[55]
Formal Specification	"a formal specification in an organizationally selected behavioral specification language (JML, Dafny, Spec#, etc.)"	[49]
Supporting artifacts		
Requirements Document	"detailed requirements document"	[47]
	"requirements"	[47]
	"Product Requirements Document"	[9]
Rationale and Intent	"declares intent"	[47]
	"the rationale (WHY) alongside the constraint (WHAT) and pattern (HOW)"	[40]
	"explicit reasoning traces"	[63]
Design Artifacts	"design support artifacts such as UML diagrams"	[49]
	"narrative description"	[49]
	"Design: In my attempt, the design document consisted of the sections seen in the screenshot below."	[9]
Implementation Plan / Task Decomposition	"Implementation Plans (plan.md) detail how to build features (analogous to technical requirements) with security considerations embedded at the design level."	[40]
	"Task Definitions (tasks.md) provide atomic work items (analogous to actionable tasks) that AI-assisted generation can execute within constitutional bounds"	[40]
	"Tasks: A list of tasks that trace back to the requirement numbers, and that get some extra UI elements to run tasks one by one, and review changes per task."	[9]
Domain Language	"domain-oriented ubiquitous language"	[55]
	"to describe business intent rather than specific tech-bound implementations"	[55]

C Mapping Spec elements to IEEE 830 SRS

Table C.1: Mapping of literature elements to IEEE 830 as Direct, Partial or No match.

Specification element	Direct	Partial	None	IEEE 830 counterpart	Notes
descriptive specifications					
behavior	X			§5.3.2 Functions	IEEE 830 uses "shall" statements; the literature describes intended system behavior.
functionality	X			§5.3.2 Functions; §5.2.2 Product functions	Both capture what the system should do.
user stories with acceptance criteria		X		§5.2.3 User characteristics; §5.3.2 Functions; §4.3.3 Verifiable; §5.3.7.4 Feature template	Content covered across sections (user roles, functions, verifiability, features) but the unified "As a... I want... So that..." construct does not exist.
scenarios (BDD / Given-When-Then)		X		§5.3.7.4 Feature; §5.3.7.5 Stimulus; §4.3.2.3	Supports stimulus-response organization but does not use Given/When/Then notation.
business rules		X		§5.3.2 Functions	No separate section; subsumed under functional requirements.
edge cases and error conditions	X			§5.3.2(c); §4.3.3(b)	Explicitly requires specification of abnormal/invalid input handling.
input/output mappings	X			§5.3.1 External interfaces; §5.3.2(e)	Explicitly requires detailed input/output descriptions.
Formal specifications					
constraints	X			§5.2.4 Constraints; §5.3.5 Design constraints	Covers regulatory, hardware, interface, and safety constraints.
invariants			X	—	Not included as a concept in IEEE 830.
preconditions and postconditions			X	—	No formal pre/postcondition language. Closest: §5.3.2(a) validity checks, §5.3.2(e) conversion formulas.
interface descriptions and integration contracts	X			§5.2.1 Product perspective; §5.3.1 External interfaces	More detailed interface requirements than the literature.
state machines / sequential logic		X		§4.3.2.3 Representation tools; §5.3.7.1 System mode	Acknowledged as representation tool; mode-based template available but not required.
formal specification		X		§4.3.2.2 Requirements specification languages	Acknowledged as option to reduce ambiguity but not required.
Supporting artifacts					
requirements document	X			§5 Specific requirements	Both organize requirements into structured documents.
rationale and intent		X		§5.2.5 Assumptions and dependencies; IEEE/EIA 12207.1 B.3(u)	Does not require rationale per requirement. Later 12207.1 standard adds it.

Continued on next page

Specification element	**Direct**	**Partial**	**None**	**IEEE 830 counterpart**	**Notes**
design artifacts		X		§5.2.2; §4.7; §4.3.2.3	Permits graphical models and narratives for *external behavior*, not internal design.
implementation plan / task decomposition			X	§4.8 explicitly excludes	Conflict: "The SRS should address the software product, not the process of producing the software product."
domain language		X		§5.1.3 Definitions; §4.3.2 Unambiguous	Requires glossary and consistency, but not ubiquitous language as a design principle.

D Data product technical components

Table D.1: Evidence traceability: data product components mapped to source quotes

Data product component	Direct Quote	Source
Data	"data within a data product can be from different types (e.g., batch files and events)"	[38]
	"a data product can be considered as combination of data..."	[30]
	"A data product is a unit of logical architecture which comprises and controls several structural components such as data..."	[60]
	"data and code"	[26]
	"the data"	[53]
	"They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage"	[8]
Metadata	"there must be an association between the data and the respective metadata"	[38]
	"rich metadata and semantic documentation"	[53]
	"a data product can be considered as combination of ... metadata..."	[30]
	"They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage"	[8]
Code	"code component encompasses three distinct segments: data pipeline, applications that allow access to the data and metadata, and the code used for access policies"	[38]
	"A data product is a unit of logical architecture which comprises and controls several structural components such as ... code, ..."	[60]
	"data and code"	[26]
	"They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage"	[8]
Transformation logic	"data may be sourced from one or more data domains and put through a variety of transformations"	[53]
	"a data product can be considered as combination of ... transformation logic, ..."	[30]
Infrastructure	"infrastructure component allows access to the data and metadata, as well as running the code related to the data product"	[38]
	"A data product is a unit of logical architecture which comprises and controls several structural components such as ... infrastructure dependencies ..."	[60]
	"data pipelines provide the technical infrastructure"	[53]
	"They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage"	[8]
Policies	"policies can be established for each data product on a detailed level"	[60]
	"A data product is a unit of logical architecture which comprises and controls several structural components such as ... policies ..."	[60]

	”They contain data, metadata, code, policies, and infrastructure dependencies to facilitate data sharing and usage”	[8]
Data quality	“Ensures the second phase of compliance and Data Quality checks”	[30]
	“a data product can be considered as combination of ... data quality, ...”	[30]
	“Trustworthiness encompasses data quality”	[53]
	“[data] quality”	[26]
SLA/SLO management	“implementation of explicit SLOs for data products”	[53]
	“a data product can be considered as combination of ... SLA management, ...”	[30]
Identity & Access Mechanism	“a data product can be considered as combination of ... identity and access mechanism ...”	[30]
	“access control”	[26]
	“a key requirement to clarify who owns and is ultimately responsible for the data products”	[26]
Incident Management	“a data product can be considered as combination of ... incident management ...”	[30]

E Data product case

E.1 Synthetic case

Create a Taxi Trip Reporting data product

As a Product Owner, I want a reusable data product based on New York taxi trip data, so that business users can analyse taxi activity in a Power BI dashboard.

Business context: Business users want to understand taxi trip patterns over time. They are interested in basic reporting on trip volume, trip duration, distance, fare amounts, and activity by pickup and drop-off location.

Available data: Use the Databricks sample data for New York taxi trips, especially the nyctaxi sample dataset where appropriate. The Product Owner does not prescribe the technical solution. The data engineer should determine which tables and fields are useful, how the data should be structured, and what transformations are needed.

Expected outcome: A data product that can support a Power BI-style reporting layer.

Business questions the data product should support:

1. How many taxi trips occur per day, week, or month?
2. What are the average trip distance and trip duration?
3. What are the total and average fare amounts?
4. Which pickup and drop-off locations have the most activity?
5. How do trip patterns differ by time of day?

E.2 Controlled-evaluation case

Instructions during evaluation

1. In github copilot, run the following prompt:
   /opsx-propose I want to create a new data product for a broker reference overview using domain_ investmentmanagementclients and domain_ clients.country.
2. Answer questions and review generated artefacts.

Guidance for use case during evaluation

Suggested product name: product_broker

Sources: domain_investmentmanagementclients.broker_marketcounterparty and domain_clients.country

Suggested output data: trade name, party type, legal form, name, region, EU membership

F DAB

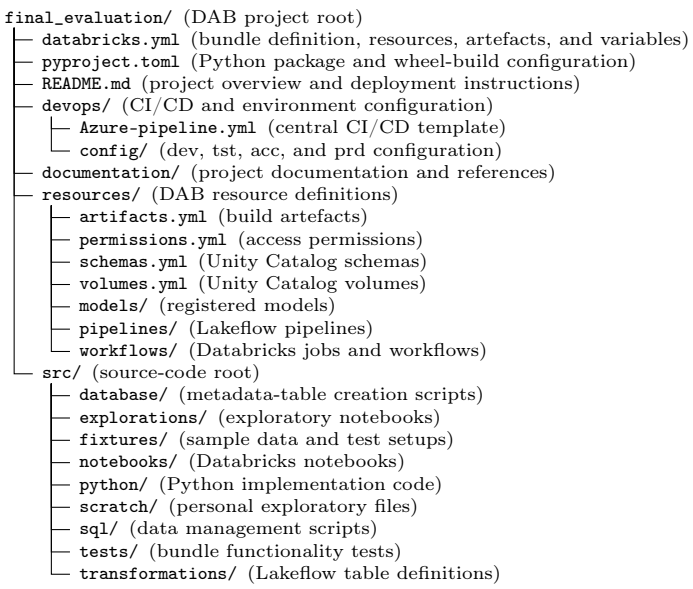


Figure F.1: Folder structure of the Declarative Automation Bundle (DAB) template

G TAM questionnaire

Table G.1: Statements included in the adapted Technology Acceptance Model questionnaire

Category	**Statement**
Perceived usefulness	Using this tool would enable me to complete data product development tasks more quickly.
Perceived usefulness	Using this tool would improve the quality of my data product development output.
Perceived usefulness	Using this tool would help me complete more data product development work within the same amount of time.
Perceived usefulness	Using this tool would help me better achieve the objectives of data product development.
Perceived usefulness	Using this tool would make data product development easier to perform.
Perceived usefulness	Overall, I would find this tool useful in data product development work.
Perceived ease of use	Learning to operate the tool would be easy for me.
Perceived ease of use	I would find it easy to get the tool to do what I want it to do.
Perceived ease of use	My interaction with the tool would be clear and understandable.
Perceived ease of use	I would find it easy to interact with this tool in the way I need.
Perceived ease of use	It would be easy for me to become skillful at using this tool.
Perceived ease of use	Overall, I would find this tool easy to use.
Willingness to use	If this tool were available in practice, I would be willing to use it in my work.

Participants rate each statement on a five-point Likert scale ranging from 1 (*strongly disagree*) to 5 (*strongly agree*).

H SDD tool analysis - generated files

File	Purpose / contents	Lines
`constitution.md`	Constitution for the data product, including five main principles. The principles appeared related to the PBI but were partly elaborated by the tool.	118
`spec.md`	Feature specification containing four user stories with priority rationale, independent tests, and acceptance scenarios. Also includes edge cases, ten functional requirements, key entities, success criteria, and assumptions.	235
`requirements.md`	Checklist covering content quality, requirement completeness, and feature readiness. All checks appeared to be marked as passed.	Not specified
`research.md`	Research notes on Databricks sample data, including table and column information, data types, descriptions, six decisions with rationales and alternatives, and resolved clarifications.	165
`data-model.md`	Data model documentation covering bronze, silver, and gold layers, entity relationships, and business rules.	244
`reporting-schema.md`	Contract for the gold/reporting layer, including catalog and schema overview, gold tables, versioning policy, IAM, refresh schedule, and metadata.	180
`quickstart.md`	Onboarding guide for a data engineer starting work on the feature for the first time.	170
`plan.md`	Implementation plan covering technical context, constitution check, project structure, research and design summaries, implementation approach, complexity tracking, and dependencies.	184
`copilot-instructions.md`	Updated with project or tool-specific instructions. Exact content changes were not recorded.	Not specified
`task.md`	Task list with 21 tasks across seven phases: setup, foundational, US1 P1, US2 P2, US3 P3, US4 P4, and polish.	167
DLT pipeline files	Implementation files for the Delta Live Tables pipeline. These were edited after execution errors until the pipeline ran successfully. Exact filenames were not recorded.	Not specified

Table H.1: Files created by Spec-kit

File	Purpose / contents	Lines
`docs/vision.md`	Product vision file containing the PBI description and describing what the product should be.	Not specified
`requirements.md`	Requirements document containing nine functional requirements, four non-functional requirements, and four constraints.	42
`entity-model.md`	Entity model document containing an entity relationship diagram, tables, attributes, descriptions, data types, and validation rules.	82
`use_cases.puml`	PlantUML use-case diagram with three actors and nine use cases.	33
Nine use-case `.md` files	One Markdown file per use case. Each file contained preconditions, a main success scenario, alternative flows, postconditions, and business rules.	Average of 60 per file

Table H.2: Files created by AIUP

File	Purpose / contents	Lines
`specs/spec.md`	Specification created after explicitly using the `requirements-gathering` and `spec-writer` skills. Includes purpose, intended users, source data, medallion architecture, supported business questions, data quality expectations, incremental load behaviour, ownership and access, and acceptance criteria.	127
Implementation files	Files created or modified when the user asked Tessl to implement the specification. Exact filenames and contents were not specified.	Not specified

Table H.3: Files created or modified by Tessl

File	Purpose / contents	Lines
`docs/technology/stack.md`	Output of the codebase-scanner skill. Describes runtime and language, platform, application structure, CI/CD, DTAP environments, and directory structure.	85
`docs/technology/dependencies.md`	Output of the dependency-inventory skill. Describes Python runtime dependencies, build and development dependencies, tooling dependencies, Databricks runtime dependencies, CI/CD external dependencies, dependency relationships, and lock file status.	74
`docs/architecture/overview.md`	Output of the architecture mapper skill. Describes system type, high-level architecture using Mermaid, data flow, deployment architecture, and bundle variables.	106
`docs/architecture/components.md`	Output of the architecture mapper skill. Describes the component map, workflow components, reserved components, and integration points.	63
`empire-test-sdd.md`	Output of the API extractor skill. Describes interface type, job parameter contracts, notebook interface, Unity Catalog schema contract, and external service dependencies.	62
`docs/architecture/data-model.md`	Output of the data model extractor skill. Describes Unity Catalog structure, ERD, configured data structures, Azure storage, Azure speech services, and data model maturity.	70
`coverage.md`	Output of the test-discovery skill. Describes test framework detection, test inventory, coverage metrics, test infrastructure, and test-to-feature mapping.	70
`prd.md`	Product requirements document. Initially generated from the DAB template because the PBI had not yet been provided, then updated after the user provided the PBI. Includes product flow diagram, product vision, user personas, feature list, non-functional requirements, and out-of-scope items.	111
Six `FRD.md` files	Six feature requirements documents, each for a different feature area. Each includes a description, user stories, functional requirements, non-functional requirements, dependencies, and current implementation. Initially generated from the DAB template, then updated after the PBI was provided.	Not specified
`adr-001-testability-gate.md`	Architecture decision record for the testability gate. Explains the testability tracks, records the hybrid choice, describes feature-level track decisions, and includes positive and negative consequences and references.	97
`increment-plan.md`	Increment plan created by the extension planner skill. Shows delivery order, five increments with track, FRD scope, complexity, dependencies, and a summary table.	203

Table H.4: Files created by Spec2Cloud

File	Purpose / contents	Lines
`proposal.md`	Proposal describing why the product is needed, what the changes are, capabilities, and expected impact.	30
`design.md`	Design document containing context, goals, non-goals, five decisions with alternatives considered, and risks or trade-offs.	59
`spec.md` file 1	Specification for one table. Includes requirements, with each requirement containing two scenarios using `WHEN` and `THEN`.	57
`spec.md` file 2	Specification for another table. Includes requirements, with each requirement containing two scenarios using `WHEN` and `THEN`.	68
`tasks.md`	Task list with 22 tasks across seven groups: setup, silver notebook, gold notebook, workflow, permissions, testing, and documentation.	48
Implementation files	Files created or modified during the apply implementation step. Exact filenames and contents were not specified.	Not specified

Table H.5: Files created or modified by OpenSpec