Friedrich-Alexander-Universität
Faculty of Engineering

Doctoral thesis

2025

Philip Heltweg, M.Sc.

Open Collaborative Data Engineering With Subject-Matter Experts Using Domain-Specific Languages

Open Collaborative Data Engineering With Subject-Matter Experts Using Domain-Specific Languages

Offen kollaboratives Data Engineering mit Fachexperten durch domänenspezifische Sprachen

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.
vorgelegt von

Philip Heltweg, M.Sc.

Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung:	16.09.2025
Gutachter:	Prof. Dr. Dirk Riehle
	Prof. Dr. Ben Hermann

Abstract

Open collaborative workflows are common, for example, in open-source software development or Wikipedia. They reduce costs for individual participants and can improve the overall quality of the result. A potential application domain for open collaboration is data engineering, especially for open data, which shares many qualities with open-source software as it can be freely used, modified, and shared.

However, data from complex domains requires the expertise of human subject-matter experts to be understood and made usable for later applications. These experts often lack the technical background needed to collaborate with software engineers using existing, text-based collaboration tools like general-purpose programming languages and project forges. Instead, various visual programming tools exist that allow non-technical contributors to build data pipelines. These tools are often proprietary and are not easy to collaborate on.

In this thesis, we explore a potential middle ground in the form of using domain-specific languages as the foundation for a shared collaboration artifact to describe data pipelines. To do so, we follow a design science methodology to identify underlying problems for collaborative data engineering with subject-matter experts, contribute an innovative artifact in the form of a domain-specific language, and empirically validate and evaluate this artifact to investigate the underlying reasons for its performance.

Initially, we summarize the literature on collaboration systems in open collaborative data engineering using a systematic literature review to develop an understanding of the current state of the art. We find a diverse ecosystem of participants, activities, tools used, and artifacts that are created during collaboration.

Based on an interview study with data engineering practitioners, we describe how their work is organized in social systems based on roles and their interactions for small-scale project groups and the wider open data ecosystem. We identify concrete challenges to collaborative data engineering and develop recommendations for resolving them.

Following up on a recommendation for the most pressing challenges, such as high technical barriers to contribution and no standard collaboration artifacts, we suggest and implement a textual domain-specific language for creating data pipelines, based on the well-known pipes-and-filters architecture.

Lastly, in a series of empirical studies with human participants, we first validate that the domain-specific language is a potential basis for a collaboration artifact for non-professional programmers and then evaluate its performance compared to Python using controlled experiments. By combining the results of the controlled experiments with a follow-up survey, we describe the effects that using a domain-specific language for data engineering has on collaborators.

Zusammenfassung

Offen kollaborative Workflows sind weit verbreitet, beispielsweise in der Programmierung von Open-Source Software oder bei Wikipedia. Sie reduzieren die Kosten für einzelne Teilnehmer und können die Qualität der Ergebnisse verbessern. Ein potenzieller Anwendungsbereich für offene Zusammenarbeit ist das Data Engineering, insbesondere für offene Daten. Diese teilen viele Eigenschaften mit Open-Source-Software, da sie frei genutzt, verändert und geteilt werden können.

Daten aus komplexen Domänen erfordern jedoch das Wissen menschlicher Experten, um sie zu verstehen und für spätere Anwendungen nutzbar zu machen. Diesen Experten fehlt oft der technische Hintergrund, um mit Softwareentwicklern über existierende, textbasierte Kollaborationstools wie Entwicklungsplattformen und Allzweck-Programmiersprachen zusammenzuarbeiten. Stattdessen gibt es verschiedene visuelle Programmiertools, die es auch nichttechnischen Teilnehmern ermöglichen, Datenpipelines zu erstellen. Diese Tools sind oft proprietär und nicht für eine einfache Zusammenarbeit gedacht.

In dieser Thesis untersuchen wir einen möglichen Mittelweg durch die Verwendung domänenspezifischer Sprachen als Kollaborationsartefakt zur Beschreibung von Datenpipelines. Dazu nutzen wir die Design-Science-Methode, um die grundlegenden Probleme des kollaborativen Data Engineering mit Experten zu identifizieren, ein innovatives Artefakt in Form einer domänenspezifischen Sprache zu entwickeln und dieses Artefakt empirisch zu validieren und zu evaluieren, um die Gründe für seine Leistung zu beschreiben.

Zunächst fassen wir die Literatur zu Kollaborationssystemen im offenen kollaborativen Data Engineering mithilfe einer systematischen Literaturrecherche zusammen, um ein Verständnis für den aktuellen Stand der Technik zu entwickeln. Wir beschreiben ein vielfältiges Ökosystem aus Teilnehmern, Aktivitäten, verwendeten Tools und Artefakten, die während der Zusammenarbeit erstellt werden.

Basierend auf einer anschließenden Interviewstudie mit Data Engineering Anwendern aus der Praxis beschreiben wir, wie ihre Arbeit in sozialen Systemen, basierend auf Rollen und Interaktionen für kleine Projektgruppen und das breitere Open-Data-Ökosystem, organisiert ist. Wir identifizieren konkrete Herausforderungen des kollaborativen Data Engineering und entwickeln Empfehlungen zu deren Lösung.

Ausgehend von Empfehlungen für die dringendsten Herausforderungen, wie zum Beispiel hohe technische Hürden zur Teilnahme und das Fehlen eines Standards für Kollaborationsartefakte, schlagen wir eine textuelle domänenspezifische Sprache zur Erstellung von Datenpipelines vor und implementieren diese.

Abschließend validieren wir in einer Reihe empirischer Studien mit menschlichen Teilnehmern zunächst, dass die domänenspezifische Sprache ein potenzielles Kollaborationsartefakt für Experten ohne Erfahrung in der professionellen Softwareentwicklung ist, und bewerten die Angemessenheit der Sprache im Vergleich zu Python mithilfe von kontrollierten Experimenten. Indem wir die Ergebnisse der kontrollierten Experimente mit einer deskriptiven Umfrage kombinieren, beschreiben wir die Effekte der Verwendung einer domänenspezifischen Sprache für das Data Engineering.

Contents

List of Figures

List of Tables

List of Acronyms

AST	Abstract Syntax Tree
DAG	Directed Acyclical Graph
DOI	Digital Object Identifier
DSL	Domain-Specific Language
GPL	General-Purpose Programming Language
GTFS	General Transit Feed Specification
IDE	Integrated Development Environment
LSP	Language Server Protocol
MADE	Methods of Advanced Data Engineering
OCDE	Open Collaborative Data Engineering
RFC	Request for Comments
RQ	Research Question
SLR	Systematic Literature Review
SME	Subject-Matter Expert
SWC	Software Campus

Acknowledgments

I would like to thank my advisor, Prof. Dirk Riehle, for guiding me through the process of becoming a researcher and for his support. I appreciate the freedom and trust shown to me, everything I could learn, and the great collaboration. Additionally, I would like to express my gratitude to Prof. Ben Hermann for his kind feedback during my research.

Everyone at the Professorship for Open-Source Software has been an amazing support. The JValue team, with Georg, Felix, and Johannes, was fun to work with and helped me immensely. Our peer debriefing group of Stefan, Georg, and Julian H. was always great with questions about research or life, and a time I looked forward to. Additionally, Julia, Nik, Martin, Thomas, Elçin, Andreas, and Julian L., thank you for always being there to help, discuss, and share ideas.

I would also like to thank my parents, Mechthild and Hermann-Josef, for their support and encouragement throughout my life. You have always supported me, and I am forever grateful.

Finally, thank you, Caro, for being there for the good days and sticking with me during the bad days. My thesis would not exist without you, and my life is better because of you. Eu te amo.

Related Publications

In this thesis, the first-person plural (*"we"*) is used as it is cumulative work, and all published papers were created in co-authorship with other researchers. For all articles, the author of the thesis was the first author; however, all co-authors provided invaluable contributions, guidance, and feedback.

This thesis includes work that was published in the following manuscripts:

1. Challenges to Open Collaborative Data Engineering [16], see Appendix A.
2. A Systematic Analysis of Problems in Open Collaborative Data Engineering [17], see Appendix B.
3. An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures [20], see Appendix C.
4. Can a domain-specific language improve program structure comprehension of data pipelines? A mixed-methods study. [19], see Appendix D.
5. Is spreadsheet syntax better than numeric indexing for cell selection? [18], see Appendix E

1 Introduction

High-quality data is the foundation for many decisions, as well as innovative software applications and artificial intelligence products. Without access to appropriate data, these projects can not be completed or are reduced in effectiveness.

An increasingly important source of data is open data, which is data published under a permissive license that can be freely used, modified, and shared by anyone, for any purpose. These datasets are mainly published by public administrations and companies due to legal requirements, but a large ecosystem of private data publishers and users exists as well.

However, most data is hard to use due to a variety of challenges, including technical ones. This is especially true for open data, which is often published out of legal necessity and not reused by the original publisher, meaning the usability of the data is a low priority, and publishers have few incentives to improve it. Data engineering, the process of extracting, transforming, and cleaning data to make it usable for a specific use-case, is therefore a costly but required engineering activity [48, 55].

Because open data shares many attributes with open-source software, data users could col-

laboratively improve the general quality of available open data in open collaborative workflows in which everyone can participate without an externally enforced process, similar to the ones found in open-source software development. Open collaborative work has proven to reduce individual workload and to improve the quality of outcomes in domains such as software engineering or the creation of knowledge bases such as Wikipedia.

Similarly, improvements in data quality or availability that benefit everyone in the ecosystem exist in open data contexts as well. As part of their larger role, infomediaries are actors that are neither data publishers nor consumers, but rather increase the use of open data. They do so by improving its supply and quality as well as building relationships [52], republishing improved datasets that make it easier for data consumers to use them in their individual projects. Therefore, on a community level, open collaborative data engineering workflows by individual open data users have the potential to improve data quality for every participant, but are, compared to software engineering, rare.

This thesis focuses on understanding collaboration systems of individual data users, what challenges they face, and how open collaboration could be further enabled in order to improve data quality for everyone. In the process of our research, we recognized the importance of subject-matter experts (SMEs) and other non-technical users as contributors to collaborative data engineering to improve the understanding of data content. We found that a missing standard collaboration artifact and inappropriate tools are some of the major problems for collaboration between subject-matter experts and software engineers. Proven tools and artifacts from open-source software engineering are often hard to understand and use for participants without a technical background, while easy-to-use visual tools are hard to collaborate with and scale poorly to larger projects.

For this reason, we study textual domain-specific languages (DSLs) as a potential middle ground. Given that DSLs can be used for open collaborative data engineering, we explore important design considerations to improve their usability by subject-matter experts. The question that we ask is: Can they be the foundation for an alternative collaboration artifact

that would still allow communities to reuse existing infrastructure and workflows from open-source software development, but be more accessible to users without a strong software engineering background?

The thesis is structured as follows: Initially, we summarize existing work on open data, collaborative workflows, and DSLs in chapter 2. In chapter 3, we outline the goals of the thesis, including research questions. We also describe the overarching research design of design science that structured our work. We present our work in understanding collaboration systems in data engineering, identifying challenges to open collaboration and deriving objectives for a potential solution in chapter 4. The design and development of the proposed solution, a DSL to create data pipelines and a test bed to evolve it, is described in chapter 5 while its appropriateness is demonstrated in chapter 6. The created artifact is extensively evaluated in chapter 7 using empirical methods. Finally, our contributions are summarized in chapter 8.

2

State of the Art

In this chapter, we discuss existing work that provides the background for our contribution. We first give an overview of open data and the data engineering work required to access it in section 2.1. Collaborative work, both in software engineering and in data engineering, is discussed next in section 2.2. Finally, we present prior research on domain-specific languages and their evaluation in section 2.3.

2.1 Open Data and Data Engineering

Open data is published under an open data license, meaning it can be accessed, used, modified, and shared by anyone for any purpose[1]. The amount of, and interest in, open data increased strongly following the 2009 Open Government Directive by the United States government [48].

The open data ecosystem consists of loosely connected data publishers, legislators, facilitators to data use (often referred to as infomediaries [52]), and data consumers [68]. Research

[1] According to the Open Definition by the Open Knowledge Foundation, *https://opendefinition.org*

into open data and its ecosystem has largely focused on data publishers and less on how open data users consume data [48, 67]. Motivations to publish open data differ, ranging from governments trying to improve transparency and increase citizen engagement to private entities looking to establish a community of users around their products. Data consumers can themselves have a commercial interest, e.g., to build or enhance their own private data products, or be individual data users such as journalists or citizen activists. Increasingly, open data is also published in the form of replication packages as part of scientific research.

An important pillar of the ecosystem is open data portals that host or link to open datasets and allow consumers to search their catalogue. While open data portals of private publishers exist, most portals are run by government entities such as govdata.de[2] by the Federal Republic of Germany. On these portals, many open datasets are published in tabular formats, for example CSV or XLS files, and are smaller than 10 MB [44, 57]. Because many open data publishers are government agencies that publish data only to fulfill legal obligations, the quality of open data is often poor, and considerable effort must be expended to clean errors and make open data usable [48].

Data engineering, any activity to extract, clean, transform, and make available data for later use, is a costly part of any project involving data [55]. In long-running projects, data engineering often takes the form of creating data pipelines instead of one-off data cleaning efforts. These data pipelines can be re-run on new datasets and can automatically fix data that is updated, such as transport schedules that are regularly changed. Especially in open data contexts, building pipelines that clean data is often the only way consumers can reliably fix errors because publishers are not interested in implementing changes themselves [16].

Regarding data pipelines, various process models (such as batch and stream processing) and configurations exist. In this work, we focus on batch processing, pipelines that are executed once or in scheduled intervals, and are most relevant for open data. Batch pipelines generally consist of data extraction, transformation, and loading steps. Depending on the order of

[2] *https://www.govdata.de/*

operations, these pipelines are referred to as ETL-pipelines (Extract-Transform-Load, data is extracted, transformed, and then loaded into a final data store called a sink) or ELT-pipelines (Extract-Load-Transform, data is loaded into an intermediate data warehouse and then transformed).

2.2 Collaborative Work and Collaborative Data Engineering

Collaborative work processes are defined by the participation of multiple contributors to achieve a shared goal. Collaboration processes have been studied across a range of domains, for example, in software engineering. Common forms of collaborative work include distributed collaboration [46] or crowdsourcing approaches like hackathons [39].

Collaborative work that is egalitarian, meritocratic, and self-organizing is called open collaboration [50]. A well-known example in software engineering is open-source software development, but open collaborative workflows exist across a variety of domains and projects, such as Wikipedia. Community platforms, such as project forges, enable open collaboration by defining standard tools and artifacts to collaboration [50]. Additionally, large-scale social coding platforms such as GitHub are important to coordinate work and increase transparency about the goals of other contributors [10]. GitHub's pull-based development flow has become a standard collaborative workflow for many software engineers due to its popularity in open-source software development. Best practices from open-source development workflows have been successfully applied to software engineering in closed environments as well, known as inner source [6].

Collaborative work processes can also be found in data science projects. Studies in large corporate environments showed that data scientists work in small, very collaborative teams [66]. To collaborate, data scientists use a wide variety of tools such as Jupyter Notebooks that allow them to share pipeline code with context; however, overly complex tools can be a challenge to collaboration, especially when subject-matter experts must be included due to the complexity of the data [59].

Similar to software engineering, distributed collaboration supported by asynchronous tools like Slack and Email is a common mode of work during data engineering [7, 66]. On the other hand, open collaboration is rare in data science, with most collaboration happening only for specific projects and in small groups with diverse groups of participants. During their collaboration, participants largely develop their own tools or create reports based on data they previously cleaned [7]. However, Smith et al. [53] showed that practices from open-source software development could be successfully applied to collaborative data science workflows in the case of feature engineering for a machine learning pipeline. They found that task management, tool mismatch, evaluation of contributions, and maintaining infrastructure are the main challenges to collaborative data science efforts.

In addition to the tool mismatch and lack of specialized tools, no centralized collaboration platforms exist for collaborative data science [7]. Existing research on tools and practices for collaborative data engineering has typically focused on individual activities such as creating labeled data [49], feature engineering [54], or versioning of datasets [2]. Evaluation of these tools has shown promising results in their specific niche; however, no standard set of practices or tools has been established for the data science process as a whole.

In open data ecosystems, an additional challenge exists because entities with different goals (data publishers and data consumers) work with one dataset [68]. While open collaborative projects in open-source software development typically work on an artifact under shared control, open data consumers often have no way to directly influence the publishers of the data they are working with. Collaboration between data publishers and data consumers, together with technical difficulties, has been described as a common challenge to open data use [68, 69]. However, research into collaboration between open data users themselves is rare, even if infomediaries demonstrate that intermediate improvements to published data have value for the ecosystem [68].

2.3 Domain-Specific Languages

Domain-specific languages (DSLs) are (programming) languages that are designed using the semantics and syntax of one specific domain, instead of covering all possible domains like general-purpose programming languages (GPLs). DSLs trade a more limited feature set for being better suited for problems in the domain they cover. Well-known examples for DSLs can be found in the domains of typesetting (LaTeX) or relational data processing (SQL). Due to their focus on one particular field of application, they can reuse glossary and concepts that are known to subject-matter experts in that domain instead of general programming concepts, allowing these experts to learn and program more efficiently [26, 36]. Therefore, DSLs can increase productivity and enable more participants to contribute to a problem solution than GPLs [42].

DSLs can be categorized along numerous dimensions. While most DSLs are text-based, graphical DSLs supported by additional tools also exist. In regard to text-based DSLs, Fowler and Parsons [13] consider the distinction between internal and external DSLs. Internal DSLs build on a pre-existing language, such as Scala, and add domain concepts or adapt syntax depending on the flexibility of the host language. The public interface of domain-focused libraries can be considered an internal DSL as well, with the API of Apache Kafka for the domain of stream processing as an example. In contrast, an external DSL is a separate language that must be supported with its own tooling, but allows for a maximum of freedom during implementation of the language itself. Often, external DSLs align with syntax conventions found in related languages, but because they are not dependent on any host language, they can incorporate any conventions from the application domain as well.

Requirements and design guidelines for DSLs are described in the literature. Requirements include capturing domain concepts, providing adequate tool support, simplicity, longevity, and high quality [34]. Karsai et al. [29] gathered design guidelines for DSLs, which included defining target users and use-cases early and asking questions to align with their mental model.

To do so, DSLs should reuse the glossary of the domain and also include informal conventions that can be learned from documents such as sketches created by domain experts [61]. Including specific domain concepts in a DSL also allows subject-matter experts to work with it more quickly [22].

Another aspect that should be decided early on is if a DSL should be textual or graphical in nature [29]. For some domains, experiments have shown that textual and graphical approaches are possible, but textual DSLs lead to higher quality models [41].

The quality of a given DSLs can be investigated in a number of ways, but generally any evaluation has to be domain-specific [36]. Evaluations matter because ultimately the usability of a DSL drives its adoption [1]. In comparison to purely technical or theoretical evaluations, empirical studies have been rare in programming language research because they are complicated and expensive to execute [1, 45]. Instead, a large number of publications on DSLs are solution proposals without empirical evaluations [35, 45]. However, researchers who complete them report that they lead to deeper insights that can not be found with other evaluation methods [5].

Empirical studies with human participants, such as controlled experiments, should already be considered during the development process to guide the design of the language [1]. However, controlled experiments have been relatively rare in the wider area of software engineering research as well [33, 58].

Nevertheless, empirical evaluations of DSLs exist in multiple application domains.

Due to their limited availability, studies with real subject-matter experts are rare. In the domain of marine science, Johanson and Hasselbring [26] compared program understanding by ecologists using a DSL with the GPL C++ and found they could complete tasks in less time and with higher correctness.

In previous studies, student participants were used as proxies for subject-matter experts. Kosar et al. compared GPLs with DSLs and appropriate libraries across a number of domains, such as GUI programming, and found that the DSLs are more accurate and efficient [36, 37,

38]. Other studies in the domains of traffic simulation and optimization [21] as well as rules for type inference [32] showed similar improvements.

Aside from controlled experiments, other empirical research methods, such as interviews with subject-matter experts, are used to evaluate challenges to the use of DSLs, such as low-quality tooling [23].

Because data engineering and working with pipelines are foundational processes in many domains, a number of DSLs that are related to the work presented here have been suggested. Closely related to working with data pipelines is the definition of scientific workflows. Workflow management systems are used to define and run pipelines of tools that transform and analyze data with high transparency and reproducibility [40]. Multiple DSLs that support these definitions exist, for example, the Common Workflow Language (CWL) [9]. Using CWL, scientists can use container technologies to define workflows consisting of any command-line enabled tool.

An example of a DSL for data pipelines in biological sciences is BigDataScript [8], created for use by subject-matter experts and using a script-style programming model. Its goal is to be architecture-independent and allow data pipelines to be executed on many different computing environments. The concept of pipelines can also be found in the domain of software engineering, for example, during continuous integration. Fonseca et al. [12] described an external DSL called PACE that improves on the previous manual editing of JSON configs by professional software engineers in an industrial context. Lastly, PiCo is a DSL using pipes and the data flow computational model. Misale [43] demonstrated and evaluated the performance of their implementation.

As Kosar et al. [36] point out, the design and evaluation of DSLs have to be domain-specific. In this thesis, we contribute to the body of knowledge on DSLs by studying the impact of design choices for DSLs in the domain of data engineering.

3

Aim and Structure of the Thesis

In this chapter, we present the main goals and research questions that are discussed in this thesis. In addition, we outline the overall structure that we followed during the research and how it maps to the published articles and chapters in this thesis.

3.1 Goals and Research Questions

Our main goal was to find out why open collaborative workflows are not as prevalent in data engineering for open data as they are in software development, and how to enable the open data community to work more collaboratively.

We focused here on a subset of identified problems, namely a shared collaboration artifact between software engineers and subject-matter experts, and appropriate tooling for it. As part of the solution, a DSL was implemented as an open-source software artifact. We cover the research and scientific contributions surrounding the language design and the creation of a test bed for continuous improvement of this language using rigorous, empirical evaluations.

Therefore, even though we present the language as part of chapter 5, we aimed to investigate

the wider context of problems in open collaborative data engineering and generalized insights into how text-based DSLs for data engineering can best be adapted to the needs of subject-matter experts.

In this thesis, we initially set out to answer one overarching research question (RQ):

Research Question 1 (RQ1): "How can open collaborative data engineering with open data be enabled?"

After investigating existing challenges and narrowing down the scope of the work to enabling subject-matter experts to contribute to collaborative data engineering projects, we additionally considered the following research questions:

Research Question 2 (RQ2): "Is a text-based DSL a viable foundation for a collaboration artifact in open collaborative data engineering?"

Research Question 3 (RQ3): "What are important considerations for a DSL to create data pipelines by subject-matter experts?"

3.2 Thesis Structure: Design Science

In this section, we discuss the thesis structure, the overarching research design that we used, and how it maps to the publications. Part of the work was completed in collaboration with an industry partner during the Software Campus project, which is presented first. Our research methodology was based on design science according to Peffers et al. [47]. We present design science, its activities, and how they relate to the research projects that we conducted. An explanation of the individual research methods and their results is discussed in the following chapters when appropriate.

An overview of the projects that are discussed here is shown in Table 3.1.

Industry Collaboration: The Software Campus

The work presented here was partially completed as part of the Software Campus (SWC). The Software Campus is an accelerator for future leaders in IT and provides funding for doctoral

Project	Methods	Publication	Results	Described In
P1	Systematic Literature Review [30] Descriptive Data Synthesis [30]	[16]	1. Overview of participants, activities, tools used and artifacts created during collaborative data engineering 2. Description of first challenges to open collaboration	Section 4.2
P2	Qualitative Survey [24] Descriptive Data Synthesis [30]	[17]	1. Identified challenges to open collaboration 2. Guidelines to enable open collaboration	Section 4.3
P3	Descriptive Survey [31] Thematic Analysis [4]	[20]	1. Viability of a DSL for data engineering by SMEs 2. Description of major effects when using a DSL for data engineering 3. Important hypotheses to test	Section 7.2
P4	Controlled Experiment [33] Descriptive Survey [31] Thematic Analysis [4]	[19]	1. Effects of a DSL on data pipeline structure understanding 2. Descriptions of reasons for the effects	Section 7.3
P5	Controlled Experiment [33]	[18]	1. Effects of spreadsheet syntax on DSL efficiency	Section 7.4

Table 3.1: Overview of projects, methods, and publications that were completed as part of the thesis.

students in computer science to complete a research project with an industry partner.

For this project, *JValue-OCDE-Case1*, we cooperated with Springer Materials, a company of the Holtzbrinck Publishing Group. Springer Materials offers a data product consisting of curated data for materials science. In addition to closed data sources, ingesting high-quality open data (for example, from scientific publications) adds additional value to their business. As a partner in the Software Campus project, they offered feedback and guidance from an industry viewpoint. We worked with industry contacts for the problem identification, and during the demonstration activity of design science research.

Design Science

Design science, according to Peffers et al. [47], is a research methodology based on the iterative creation and evaluation of an innovative artifact to solve a real-world problem. Six main activities are part of the design science process: Problem identification, objective definition, design and development, demonstration, evaluation, and communication. While these activities are presented as a linear process, in practice, they are iterative, and researchers can revisit earlier activities if new information requires an adaptation.

1. **Problem identification**: During problem identification, researchers define a specific research problem and motivate its solution. Concluding this activity requires a clear understanding of the problem domain that has to be developed. We approached the problem identification phase by first familiarizing ourselves with the academic literature on collaboration systems in data engineering by open data users using a systematic literature review (see subsection 4.1.1). Building on the knowledge gained from the literature, we extended our problem identification with a qualitative survey using semi-structured interviews with open data practitioners (see subsection 4.1.2).

2. **Objective definition**: With a deeper understanding of the problems in the field of research, the researchers define objectives for a solution, based on what is considered possible to build and which alternative solutions already exist. Objectives also lay the basis for how the solution is evaluated during the following activities. In our work, we derived the objectives based on the challenges to collaborative data engineering we identified previously and describe them in section 4.4. To ensure we could make meaningful progress, we focused on a subset of the most important challenges that we were in a good position to propose solutions for in the form of a complex software engineering artifact.

3. **Design and development**: The design science artifact is created during the design and development activity. An artifact does not have to be physical or a software solution, but can also be a new model, theory, or even an improvement for already existing solutions. In our work, the developed artifact is an open-source DSL, called Jayvee, to build data pipelines and the accompanying infrastructure to empirically evaluate and improve it using scientific methods. We give an overview of the language design, technology choices, and the development process in chapter 5.

4. **Demonstration**: After it is developed, the artifact is used to solve at least one instance of the described problem to demonstrate that it is usable and appropriate. In contrast to the following evaluation, the goal of the demonstration is only to show that the artifact can be used to solve the problem at all, without formally evaluating the outcome. We were able to

demonstrate the use of our artifact in two different use-cases, first with students solving data engineering exercises (see section 6.2) and second with industry during the project completed as part of the Software Campus (see section 6.1).

5. Evaluation: During evaluation, the artifact is observed while it is being used to solve the problem, and the outcomes are measured. These performance insights are then compared against the objectives defined earlier. Depending on the goal of the evaluation, different evaluation methods, measurements, and analyses can be appropriate.

For our evaluation, we followed a two-step process. First, we evaluated if the proposed DSL was a potential basis for a collaboration artifact for data engineering and developed potential hypotheses to be evaluated by working with user feedback using qualitative methods (see section 7.2). Second, we narrowed our focus and evaluated concrete hypotheses using mixed-methods (in section 7.3) and later with a controlled experiment (see section 7.4). To do so, we created a module called *Methods of Advanced Data Engineering* to teach students about open data and data engineering while introducing them to Jayvee (described in more detail in section 6.2). The module enabled us to evaluate the language with feedback from a large number of users, while also providing a test bed for future language design based on empirical data.

6. Communication: Finally, the knowledge about the identified problem, the proposed solution in the form of the artifact, and the results of the evaluation must be communicated to the broader scientific community. As a cumulative thesis, the scientific contributions of the work described here have been continuously communicated in the form of scientific publications. The overall context and process are additionally described in this thesis. Because design science is an iterative process, multiple cycles of objective definition, design, development, and evaluation are typically completed during a project, while the final report presents only the outcome of the work linearly. This is also the case in this thesis, where we will focus on the final result of the process instead of reporting on each iteration.

To summarize, Figure 3.1 shows an overview of the activities and how they relate to the

research projects we conducted.

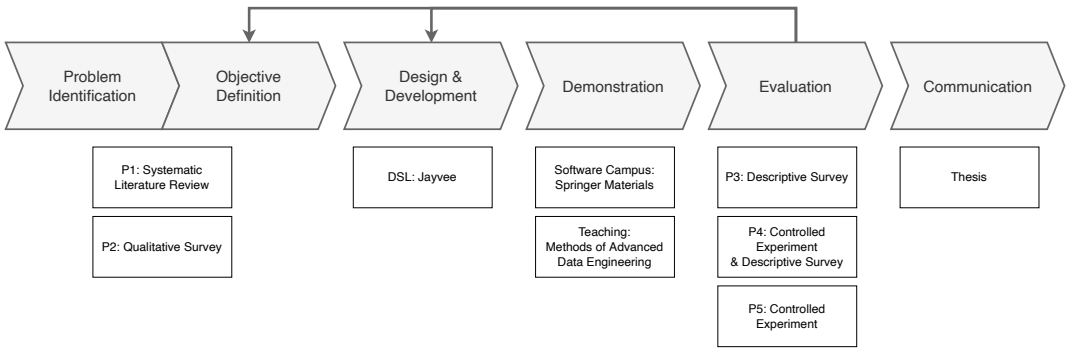


Figure 3.1: Activities of the design science process according to Peffers et al. [47] with associated projects and artifacts described in this thesis.

Each of the following sections will focus on a specific activity of the design science process, from problem identification to evaluation, highlighting the research methods used and main results, as well as how they were communicated in scientific publications.

4

Problem Identification and Objective Definition

During the problem identification and objective definition, we started with explorative research studies to build an understanding of major challenges to open collaborative data engineering. Initially, we took into account a diverse set of sources and projects. In addition to providing a wider overview of the field, including multiple viewpoints, such as academic literature and interviews with data practitioners, also allowed us to investigate the same challenges from different angles.

In this chapter, we first present the research methods we used in section 4.1, followed by major research projects we conducted during problem identification and their main results (section 4.2 and section 4.3). Finally, as a result of the activity, we derived objectives from the identified problems that guide the design and development of the design science artifact in section 4.4.

4.1 Research Methods

During problem identification, we used systematic literature reviews according to Kitchenham [30] to familiarize ourselves with the existing academic literature and prepared a qualitative survey according to Jansen [24] to gain insights from practitioners working with data.

4.1.1 Systematic Literature Review

Systematic literature reviews according to Kitchenham [30] can be used to understand the current state of the art in the academic literature. The steps are first to plan the review, then to conduct the review (including selecting primary studies and extracting and synthesizing data), and finally to report the review in an appropriate venue.

During planning, researchers define a research protocol based on the motivation for a systematic review. This research protocol includes the research questions that should be answered, a clearly defined search strategy with objective inclusion and exclusion criteria, and quality measurements for primary studies. Additionally, researchers should decide at this point on a strategy for data extraction as well as a plan for data synthesis.

Based on the research protocol, the structured review is then conducted. While carefully documenting the process, the researchers are executing the search for primary studies as defined previously. Found studies are evaluated based on the inclusion/exclusion criteria and their quality, after which they are either retained or removed from the pool of relevant literature.

Once relevant studies have been identified, data can be extracted and finally synthesized using appropriate research methods depending on the research questions. Kitchenham includes suggestions for purely descriptive approaches as well as quantitative synthesis using statistical methods.

4.1.2 Qualitative Survey

The qualitative survey, according to Jansen [24], is a research method appropriate for exploring the diversity of a domain under study. In contrast to a statistical survey, which describes distributions of variables in a population based on quantitative data, qualitative surveys gather qualitative data (typically using interviews) from a defined population. When performing a qualitative survey, the first step is to operationalize a way to answer the research question by defining a knowledge aim consisting of the topic to be studied (the material object), the aspect of it to study (the formal object), the empirical domain, and the unit under observation.

Regarding sampling, qualitative surveys aim for a diverse sample that covers all relevant expressions of the phenomenon under study. This sampling can either be achieved by creating a sampling model in advance (e.g., with knowledge from previous studies about the population) or by sampling iteratively, analyzing the collected data for categories, and adapting the sampling model until a predefined stopping criterion is reached.

Data collection is most often done by interviewing members of the population, either using free-form or semi-structured interviews and transcribing them. It is also possible to use other qualitative data sources, such as meeting minutes or observation notes. After collecting data, it can be analyzed descriptively. Additionally, a variety of appropriate data analysis methods for qualitative data can be used to extract meaning from the data. Generally, a qualitative survey can be either inductive or deductive. When conducting an inductive qualitative survey, the raw data is analyzed without preconceived theory, while in a deductive study, at least some predefined structure (for example, from an existing theory) is applied when analyzing the data. Especially inductive qualitative surveys are often iterative, where new data is collected and analyzed until a stopping criterion is reached. Qualitative surveys can also be executed as a single run with a data gathering phase, followed by an analysis phase, usually when based on existing knowledge from a pre-existing theory.

For the analysis phase, Jansen describes three categories of possible analysis: Unidimensional and multidimensional description, as well as explanation. In unidimensional approaches,

data is organized into objects, their dimensions, and categories for each dimension. To accomplish this, parts of the data are assigned a code, and codes can be related to each other. In downward coding, codes get increasingly more specific to describe diversity. In upwards coding, codes become increasingly abstract to show common patterns in the data. During multidimensional description, the data is synthesized either by concept or by case. In the concept-oriented approach, common topics or categories are extracted from multiple samples, while in the case-oriented approach, multiple similar cases are grouped together based on their characteristics and assigned labels. In addition to these descriptive categories or labels, a data analysis approach in the explanation category would relate them to a wider context.

4.2 Collaboration Systems in Open Collaborative Data Engineering

As a first step to problem identification, our goal was to understand how data users collaborate during data engineering with open data. While data engineering by data publishers is a relatively well-covered topic in the literature, few studies exist that summarize how data users improve their data. We conducted an exploratory systematic literature review to gain an understanding who participates during collaborative data engineering by data users, what activities they complete, which tools they use and what artifacts they create. Here we present the main results that relate to the thesis topic, for additional details see the full publication in Appendix A Heltweg and Riehle [16].

4.2.1 Study Design

After first reading existing literature in an ad-hoc manner, we designed a systematic literature review according to Kitchenham [30]. We decided to search in Scopus and Google Scholar to include most scientific publications. For both, we designed search queries based on a combination of `open data` with `workflow`, `process`, `practice`, or `participant`, their plurals and synonyms. We included only articles that describe data engineering workflows or concrete projects with open data and excluded any articles that are not peer-reviewed, not accessible to us, or that

only present the point of view of data publishers.

Based on these searches and forward snowballing [62], we found 487 primary studies that we filtered for duplicates, study type, and relevance to create a final set of 18 relevant articles. From these articles, we extracted elements of collaboration systems (participants, activities, tools, and artifacts) in open collaborative data engineering by open data practitioners. We also noted some potential challenges to explore in follow-up studies. For data extraction, we followed the guidelines for descriptive synthesis by Kitchenham and created a data extraction form to gather the relevant elements from each article in a structured manner.

Because we were interested in the diversity of existing elements and did not want to make statistical inferences at this point, we decided to use theoretical saturation as a stopping criterion [3]. Therefore, we tracked how many new elements we identified with each processed article and considered theoretical saturation to be reached when we did not gain any new insights in the final studies.

4.2.2 Data Engineering Includes Social Activities

We described a wide range of activities that are completed as part of data engineering efforts by participants, as shown in Table 4.1.

Acquire	Assess	Communicate	Extend	Improve	Maintain	Understand
Build Infrastructure	Ensure Anonymity	Ask Publisher	Add Metadata	Aggregate	Archive	Analyze
Discover	Evaluate	Discuss	Create Features	Clean	Document	Ask Experts
Extract	Preview	Find Community	Label	Combine	Refresh	Experiment
Read Documentation	Measure Availability	Find Skilled Users	Rate	Curate		Learn Subject-matter Knowledge
Search	Verify License	Give Feedback	Translate	Enrich		Learn Structure
Select	Visualize / Plot Data	Request Data		Link		
Store		Share Data (Publisher)		Normalize		
Validate		Share Data (Stakeholders)		Reformat		
		Share Information		Repair		
				Structure		

Table 4.1: Activities performed during data engineering (adapted from [16]).

Aside from the expected technical work, such as creating software and infrastructure to extract and clean data, we observed that participants also complete activities related to understanding data or communicating with other stakeholders. In regard to understanding data, participants either engage the data directly, e.g., by experimenting with the data or exploring

its structure, or they have to learn subject-matter knowledge in the application domain of the data. The need for deeper subject-matter knowledge is especially prevalent in fields with more complex datasets, such as data from open science projects. We found that, to acquire this knowledge, activities related to communication are important. Often, data users have to communicate with data publishers to ask questions about the data offers or report issues with the data. In addition, data users try to find other users to help them either with specialist skills, such as software engineering, or subject-matter knowledge to make sense of the dataset content. In this sense, collaborative data engineering is as much a social challenge as it is a technical one.

4.2.3 Participation by Subject-Matter Experts

Participants from a diverse set of backgrounds engage in data engineering. Table 4.2 provides an overview of the roles we extracted from the literature.

Participants	
Businesses	Mediators
Citizen Scientists	NGOs
Civil Servants	Open Data Experts
Data Scientists	Organisations
Subject-matter Experts	Private Citizens
Goverment Agencies	Researchers
Hackathon Participants	Software Developers
Infomediaries	Startups/Entrepreneurs
Journalists	Students
Legal Advisors	

Table 4.2: Participants in data engineering, by user role (adapted from [16]).

Noteworthy is the participation of users with specialist knowledge, such as experts in the subject matter of the data, legal advisors, or open data experts who are familiar with the wider data ecosystem. As a result of the need to understand data, these participants collaborate with other data users to share their knowledge. In these collaborations, experts without a software development background face challenges with the high entry barriers to programming and

have to collaborate with software engineers who actually build the pipelines handling the data.

Additionally, subject-matter experts are also data users themselves and need high-quality data for their own projects, e.g., to analyze during scientific research or to build products in industry. In these cases, experts need to either reach out to other participants with specialized knowledge in software engineering or try to create data pipelines themselves.

4.2.4 Conclusion and Future Research

Based on our systematic review of the existing literature on collaborative data engineering with open data, the experiences and challenges of data users are not clearly described. In the collaborations we investigated, participants completed the expected technical work, such as building software to extract and clean data. However, many activities are of a social nature, such as reaching out to subject-matter experts to understand data content or working with legal advisors to clarify allowed usage. Participants come from a multitude of backgrounds with varying previous experiences in software engineering. As a result, users without a programming background collaborate with software engineers or face high technical challenges when completing their own data engineering projects.

With the insights from this study, we defined a follow-up qualitative survey using semi-structured interviews with data engineering practitioners to gain a more detailed, first-hand knowledge of the challenges that data users face during data engineering. Our goal was to add additional rigor to the results by data source triangulation [56] and to clearly describe the work dynamics and challenges in open collaborative data engineering.

4.3 Problems in Open Collaborative Data Engineering

Building on the knowledge gained in the systematic literature review (section 4.2), we conducted a qualitative survey using semi-structured interviews with data engineering practitioners according to Jansen [24]. In addition to verifying the elements of collaboration systems found previously in the literature, we asked for more detailed insights into the social systems

(including roles and interactions) in which participants work during data engineering, and explicitly about challenges they face when performing collaborative data engineering. The complete study has been published as part of Heltweg and Riehle [17] (see Appendix B). Here, we summarize the main findings regarding roles in collaborative data engineering project teams and challenges relevant to the research questions.

4.3.1 Study Design

We developed the qualitative survey according to Jansen [24] by defining the knowledge aims as understanding the diversity of social systems and challenges during collaborative data engineering by data engineering practitioners. Our sampling model was based on our previous insight into the population with categories relating to job role, employer, project type, and subject matter. We considered professional and hobbyist data practitioners and whether they were mainly working with open or private data. As we were still in an exploratory phase of the project and did not need to make generalized conclusions based on statistical methods at this point, we employed convenience sampling from our personal networks and industry contacts from the Software Campus project (see section 3.2). For this sampling, we reached out to interview participants from various roles, employers, and subject-matter domains that worked with open and private data.

Before starting the interviews, we designed an interview guide that allowed us to make sure interviews would contain sections about demographic data, collaborative data engineering itself, the social systems participants worked in, and the challenges they faced. We left opportunities for participants to go deeper into any topic they felt strongly about or to add further points at the end of the interviews. Similar to the systematic literature review, we used descriptive data synthesis as described by Kitchenham [30] to analyze the qualitative data we gathered when transcribing the interviews. Because of our pre-existing theory, based on the structured literature analysis described in section 4.2, we approached the data analysis as deductive. We extracted roles and interactions that existed in social systems, as well as social and technical

challenges mentioned by interviewees.

4.3.2 Roles in Collaborative Data Engineering

In the context of larger collaborative projects in the open data ecosystem, roles can be categorized by the type of involvement in a concrete data project. We observed that, at its core, a project group is directly working on a specific application based on data. A larger data community consisting of other data users, data publishers, or external subject-matter experts interacts with the project group, facilitated by individual participants who fulfill communication and connection roles. These data communities are most obvious in open data environments where they are loosely connected by a shared interest, but similar communities exist in closed data contexts as well, for example, when multiple business units work with the same dataset. Lastly, sometimes auxiliary roles such as infrastructure providers, external software developers (e.g., developing open-source software that is used by the project group), or statisticians can be observed that support the project group on very specialized tasks.

Of high interest to collaborative work in data engineering are the social dynamics inside the project group itself, as they are the core of all collaborative data engineering projects, and they work together most closely. We found that it includes contributors that fulfill the roles of software developer, creating the software artifacts that ultimately handle the data, data engineer, who provides guidance on how to work with data regardless of its type and subject-matter expert, a non-technological role that provides expert knowledge to understand the semantic content of the data. A mediator role facilitates the exchange of knowledge between these roles. Depending on the nature of the data, the skills required to fulfill any of these roles might be very advanced (with complex data, for example, from scientific measurements). It is possible for one person to combine multiple roles, for example, for a technical person with a programming background to both build software artifacts as a software developer and understand how to work with data as a data engineer.

Because of the different knowledge required to serve in these roles, some participants have

a technical viewpoint on data engineering, while others have a subject-matter viewpoint. On the one hand, contributors with technical expertise understand how to develop infrastructure for data or how to work with common file formats, but struggle to understand the meaning of complex datasets. On the other hand, subject-matter experts know the content of the data and can find and resolve semantic issues, but face technical challenges when they have to create software artifacts to work with data. Between these two viewpoints, a mediator has to explain both technology and subject matter, either by having a basic understanding of both or by developing that understanding from talking to other contributors. Without this facilitation, subject-matter experts and the more technical roles of software developer and data engineer struggle to collaborate.

Figure 4.1 shows the roles described in a project group and the social interactions related to this mediation process.

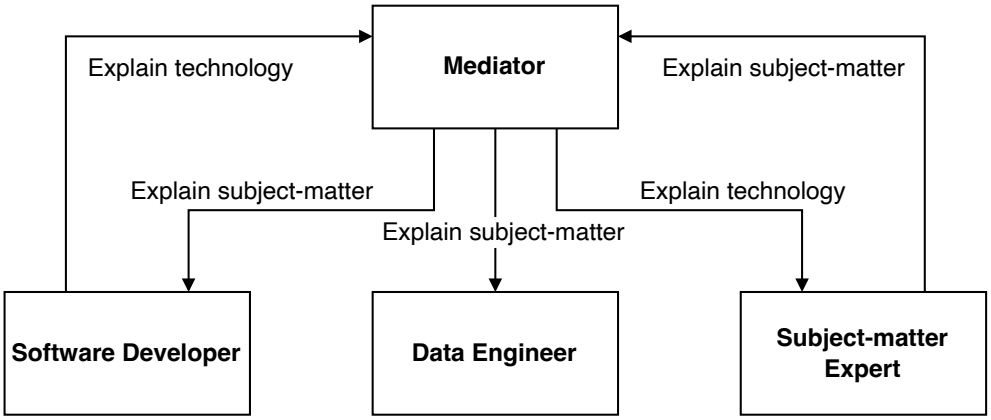


Figure 4.1: Roles and social interactions in a project group during the mediation process as part of collaborative data engineering (adapted from [17]).

4.3.3 Lack of Purpose-Built Tools and Standard Artifacts

We identified a number of both technical and social challenges faced by data practitioners during collaborative data engineering from the interviews. For completeness, an overview of all challenges, coded as C1-C13, is shown in Table 4.3.

ID	Type	Title
C1	Technical	Need for specialized skills but high barriers to participation
C2	Social	Finding and connecting with community members
C3	Social	No well-understood collaboration practices
C4	Technical	No standard tools or artifacts
C5	Technical	Data representation
C6	Technical	Inadequate tools
C7	Technical	Infrastructure for data projects
C8	Technical	Bad data sources
C9	Social	Conflicts with data publishers
C10	Social	Unclear data use cases
C11	Social	Data semantics
C12	Social	Missing incentives
C13	Social	Missing knowledge

Table 4.3: Challenges to open collaborative data engineering (adapted from [17]).

Some, such as the need for specialized skills but high barriers to entry (C1), problems with data infrastructure (C7) or semantics (C11), as well as missing knowledge (C13), directly arise due to the multiple viewpoints on data explained earlier.

In comparable open collaboration workflows such as open-source software development, technical solutions such as version control systems like git and project forges like GitHub have contributed to reducing technical challenges and enabling open collaboration. Additionally, a standard collaboration artifact exists in open-source software: Programs are expressed in text as source code, meaning collaboration tools can focus on one artifact to support, and community standards can develop for collaboration processes involving this artifact (such as the pull-request-based workflow used by GitHub).

In contrast, we could neither find well-understood collaboration practices (C3) nor standard tools or artifacts (C4) that are used during collaboration in data engineering with open data. The technical members of the community often use tools that are originally built to support software engineering (such as git and GitHub) to collaborate on source code in general-purpose programming languages (GPLs). While these tools are usable for collaborative data engineering, they do not ideally support it and, in fact, might hinder the adoption of spe-

cialized tools by being just good enough not to search for better alternatives. On the other side, subject-matter experts often rely on visual programming tools to build data pipelines or spreadsheet software to clean data. These tools are mostly proprietary and do not support collaboration well. Instead, they focus on being easy to use for individual, non-technical users. With these differences, subject-matter experts encounter high technical barriers to collaboration when they need to contribute source code in a GPL, while technical users do not know the tools subject-matter experts use and can not reuse the existing collaboration infrastructure from open-source software development.

From these insights, we created a number of guidelines to enable open collaboration during data engineering as shown in Table 4.4. They include general problems to be aware of with open data, such as low quality or non-responsive publishers (G1). Most importantly, however, we described guidelines that can be followed by project teams to enable easier collaboration, such as making sure the project is accessible to software developers, data engineers, and subject-matter experts alike (G2) and agreeing on standard collaboration practices and artifacts (G3). One way to achieve these goals is to create purpose-built tools for collaborative data engineering that can be used by technical and non-technical contributors (G4).

ID	Guideline
G1	Plan with data problems like distributed sources, updates, low-quality, and limited access to publishers
G2	Make projects accessible to data engineers, software developers, and subject-matter experts
G3	Enable collaboration by agreeing on standards, improving project visibility, and curating data
G4	Support projects with tools built specifically for collaborative data engineering

Table 4.4: Guidelines to enable open collaborative data engineering (adapted from [17]).

4.3.4 Conclusion and Next Steps

With additional data from a qualitative interview study with data engineering practitioners, we found multiple roles with varying degrees of technical expertise that collaborate during data engineering. These roles include subject-matter experts, who understand the semantic meaning of data, as well as software developers and data engineers, who can work with data on a technical level. Contributors to collaborative data engineering projects face a variety of technical and social challenges to collaboration. In contrast to open collaborative workflows that have been shown to work well, such as open-source software development, open collaborative data engineering lacks specialized tools and standard collaboration artifacts. Instead, technical contributors rely on existing open-source software development infrastructure, such as git and GitHub, while subject-matter experts use visual tools or spreadsheet software to solve their individual data engineering problems.

The domain knowledge we gained and the challenges we identified conclude the problem identification step of the design science process. We derive objectives to solve by designing an artifact in the following section.

4.4 Objectives for Design and Development

During the problem identification, we developed an understanding of the main challenges to open collaboration in data engineering (shown in Table 4.3) and the social dynamics in project groups. We described the participation by subject-matter experts and friction when working with non-optimal tools due to the high barriers to participation.

With reference to RQ1 (in section 3.1), how open collaborative data engineering can be enabled, reducing these challenges would lower barriers to entry and increase participation. Especially the lack of a standard collaboration artifact for both subject-matter experts and technical participants is an important challenge for which we were well-positioned to propose a solution. As a middle ground between text-based GPLs and visual tools, DSLs stand

out as a potential solution that could allow software engineers to reuse existing collaboration infrastructure and still be accessible to subject-matter experts.

Derived from these insights, we defined our objectives for the design and development phase:

Objective 1: Develop a text-based DSL to create data pipelines that is easier to use by subject-matter experts than a GPL.

By designing and developing a new DSL as the artifact of the design science process, we can potentially validate and evaluate it iteratively to answer RQ2, regarding the viability of a DSL for open collaborative data engineering, and RQ3, regarding the important considerations of a language for subject-matter experts, respectively.

Because our goal was to create a purpose-built language for experts in subjects we are not familiar with, we did not attempt a purely theoretical construction. Instead, we understood the design of the language as an empirical software engineering challenge in which we iteratively designed and evaluated language features based on user feedback.

Objective 2: Create a test bed for future language development to support open collaborative data engineering with subject-matter experts.

As a result of the empirical software engineering approach to language design, our goal was to create a solid foundation to get user feedback and run experiments to gather data to evaluate features. With this development approach, we were building out a theory of which features are important for a good DSL developed for subject-matter experts, answering what important considerations exist for DSLs when it comes to creating data pipelines by subject-matter experts in general (RQ3, section 3.1).

5

Design and Development

Based on the previously defined objectives, our goal was to develop a text-based DSL that makes creating data pipelines easier for subject-matter experts. Additionally, the language ecosystem should be usable as a good test bed for future language design, based on empirical data from users.

The implementation of the language as a software artifact is mainly a software engineering challenge. In the context of this thesis, the main scientific purpose of developing a new DSL was the ability to make use of it in empirical studies to guide the incremental evolution of the language and provide knowledge for similar projects. These evaluations, including the development and testing of hypotheses related to important features for DSLs used in data engineering, are described in chapter 7.

In this chapter, to provide the necessary context, we start by giving an overview of the final output of the design and development activity, the DSL Jayvee itself in section 5.1. Afterwards, we present the underlying ideas behind the initial language design in section 5.2, the process we followed while developing it (section 5.3), and major implementation choices (sec-

tion 5.4).

5.1 Jayvee, A Domain-Specific Language to Create Data Pipelines

We implemented Jayvee, a DSL to model data pipelines. Additionally, we wrote a reference interpreter that parses a Jayvee model and executes one pipeline run as a batch process. In support of the language itself, we published a language server and a VSCode plugin that provides syntax highlighting and autocompletion to edit Jayvee models. The entire project is available as open-source software on GitHub[1] under the AGPL-3.0-only license.

As previously described, Jayvee aligns closely with the mental model of data pipelines as a connected series of processing steps and follows a pipes-and-filters architecture. Jayvee models consist of *pipelines* that include these processing steps (called *blocks*). Blocks are connected in a directed, acyclic graph (DAG) with *pipes*. This structure represents a data flow from a source (a block without an input pipe that extracts data), over transformation steps (blocks with an input and output pipe), into a sink (a block with an input pipe but without an output pipe).

As an accompanying example, consider the Jayvee model given in Listing 5.1. The model describes a data pipeline that extracts open transport data, parses it, and saves it in a SQLite database. The data is provided in the General Transit Feed Specification (GTFS) format that consists of a ZIP file containing CSV files following standard schemata. The model is structured with one `pipeline` (called `GTFSPipeline`) and multiple blocks (following the `block` keyword, such as the `GTFSFeedExtractor`). The blocks are connected into a DAG with pipes (using the `->` syntax) in lines 4-10.

Jayvee models can `publish` elements from files and `use` elements that have been published in the workspace. In addition, a standard library of common block and value types is loaded automatically for every Jayvee model.

[1] *https://github.com/jvalue/jayvee*

Listing 5.1: A complete Jayvee model to extract an open transport dataset, parse it, and save the transport stops in a SQLite database.

```
use { Latitude, Longitude } from "./value-types.jv";

pipeline GtfsPipeline {
  GTFSFeedExtractor
    -> ZipArchiveInterpreter
    -> StopsFilePicker
    -> StopsTextInterpreter
    -> StopsCSVInterpreter
    -> StopsTableInterpreter
    -> StopsLoader;

  block GTFSFeedExtractor oftype HttpExtractor { url: "https://developers.google.com/static/transit/gtfs
      /examples/sample-feed.zip"; }

  block ZipArchiveInterpreter oftype ArchiveInterpreter { archiveType: "zip"; }

  block StopsFilePicker oftype FilePicker { path: "/stops.txt"; }

  block StopsTextInterpreter oftype TextFileInterpreter { }
  block StopsCSVInterpreter oftype CSVInterpreter { enclosing: '"'; }

  block StopsTableInterpreter oftype TableInterpreter {
    header: true;
    columns: [
      "stop_id" oftype text,
      "stop_name" oftype text,
      "stop_lat" oftype Latitude,
      "stop_lon" oftype Longitude,
    ];
  }

  block StopsLoader oftype SQLiteLoader { table: "stops"; file: "./gtfs.db"; }
}
```

Blocks have an `oftype` relationship to a block type, either built-in (with its execution implemented in the interpreter) or written in Jayvee itself. To describe a block type in Jayvee, a `composite` block is used that defines properties, inputs, outputs, and an internal pipeline from input ports to output ports. The Jayvee standard library includes some domain-specific composite block types, such as the `GTFSExtractor`, shown in Listing 5.2, that combines the download of a GTFS file and the extraction of the underlying ZIP archive. Similar composite blocks exist for interpreting the contents of a GTFS dataset according to the standard definition.

Listing 5.2: The GTFSExtractor, a domain-specific composite blocktype to extract GTFS data, implemented in Jayvee.

```
1  publish composite blocktype GTFSExtractor {
2    property url oftype text;
3    input inputPort oftype None;
4    output outputPort oftype FileSystem;
5
6    inputPort
7      -> FileExtractor
8      -> ZipArchiveInterpreter
9      -> outputPort;
10
11   block FileExtractor oftype HttpExtractor { url: url; }
12   block ZipArchiveInterpreter oftype ArchiveInterpreter { archiveType: "zip"; }
13 }
```

Value types in Jayvee are described by a `valuetype` element that contains a named collection of constraints that restrict valid values for built-in or user-defined value types. If a value does not conform to the defined value type, it is discarded as invalid. Value types are used in Listing 5.1 to define an expected data schema in the `StopsTableInterpreter` block (line 32). Some are built-in value types (such as `integer`), others are imported from an additional file where they are defined by the model author as shown in Listing 5.3.

Listing 5.3: Value type definitions in Jayvee using a constraint with expressions.

```
1 constraint GeographicCoordinateRange on decimal: value >= -180 and value <= 180;
2 publish valuetype Longitude oftype decimal {
3   constraints: [ GeographicCoordinateRange ];
4 }
5 publish valuetype Latitude oftype decimal {
6   constraints: [ GeographicCoordinateRange ];
7 }
```

Using Jayvee, data can be transformed with `transform` elements that use expressions written in a limited set of operators and the `value` keyword to calculate an output value from any number of input values.

Using the Jayvee interpreter, the example model shown in Listing 5.1 can be executed, optionally with added debug output using a command-line command: `jv -d model.jv`. Listing 5.4 shows the resulting output. An overview of the instantiated pipeline (lines 1 - 9) is always printed, followed by debug output (lines 11 - 18, truncated for readability).

Listing 5.4: A complete Jayvee model to extract an open transport dataset, parse it, and save the transport stops in a SQLite database.

```
[GtfsPipeline] Overview:
  Blocks (7 blocks with 1 pipes):
   -> GTFSFeedExtractor (HttpExtractor)
     -> ZipArchiveInterpreter (ArchiveInterpreter)
       -> StopsFilePicker (FilePicker)
         -> StopsTextInterpreter (TextFileInterpreter)
           -> StopsCSVInterpreter (CSVInterpreter)
             -> StopsTableInterpreter (TableInterpreter)
               -> StopsLoader (SQLiteLoader)

  [GTFSFeedExtractor] Fetching raw data from https://developers.google.com/static/transit/gtfs/examples/
       sample-feed.zip
  [GTFSFeedExtractor] Successfully fetched raw data

  // ... more debug output

  [StopsLoader] Inserting 9 row(s) into table "stops"
  [StopsLoader] The data was successfully loaded into the database
  [StopsLoader] Execution duration: 4 ms.
[GtfsPipeline] Execution duration: 258 ms.
```

The execution of the data pipeline described in the example model downloads the GTFS source, cleans the content according to the defined value types, and creates a SQLite file as sink that contains stops data, as shown in Figure 5.1.

A full documentation of Jayvee is available online[2]. Additional examples for the use of Jayvee can be found in chapter 6.

[2] *https://jvalue.github.io/jayvee*

Table: stops

	stop_id	stop_name	stop_lat	stop_lon
	Filter	Filter	Filter	Filter
1	FUR_CREEK_RES	Furnace Creek Resort (Demo)	36.425288	-117.133162
2	BEATTY_AIRPORT	Nye County Airport (Demo)	36.868446	-116.784582
3	BULLFROG	Bullfrog (Demo)	36.88108	-116.81797
4	STAGECOACH	Stagecoach Hotel & Casino (Demo)	36.915682	-116.751677
5	NADAV	North Ave / D Ave N (Demo)	36.914893	-116.76821
6	NANAA	North Ave / N A Ave (Demo)	36.914944	-116.761472
7	DADAN	Doing Ave / D Ave N (Demo)	36.909489	-116.768242
8	EMSI	E Main St / S Irving St (Demo)	36.905697	-116.76218
9	AMV	Amargosa Valley (Demo)	36.641496	-116.40094

Figure 5.1: An SQLite database file, created by a Jayvee model downloading an open transport data file. The table shown includes stop ids, names, and locations, extracted and validated from the original GTFS format.

5.2 Initial Language Design

With a background in agile software development methods, our approach to language design was iterative and to take early user feedback into account. However, we still needed an initial version of the language before we could gather information on potential improvements.

To start, we followed guidelines from academic literature on DSLs, which emphasize the importance of aligning with the mental model and notation that is used by the eventual users of the DSL and not relying too closely on terms from language theory [29, 61]. In regard to data pipelines, we understood that most data practitioners were either using visual programming tools, which presented data pipelines as a graph of connected processing steps, or thought of them that way. We also looked at informal notations of data pipelines (as suggested by Wile [61]), such as sketches and diagrams, which are also typically using a boxes and arrows notation.

Jayvee's execution semantics are defined by a reference interpreter that allows us to evolve the semantics together with the implementation under test. To arrive at a working version fast, we started out by providing a small set of basic functionality using the language structures pro-

vided by this interpreter. Following those, we bootstrapped as soon as possible into describing more complex processing steps in the language itself using a combination of these atomic steps, motivated by the well-known *composition over inheritance* guideline from object-oriented programming. This understanding closely resembles the pipes-and-filters architectural pattern used for data processing [14]. However, for users without a technical background who are not familiar with this approach, the name *filter* is not very intuitive. Because this structure invokes the idea of building a larger structure from small blocks, we ultimately decided to call these processing elements *blocks* and connectors *pipes*. In order to keep the initial version as simple as possible, we decided on a minimal set of language elements, adding only a `pipeline` element to group blocks and pipes.

In addition to the design considerations regarding the language, we understood from our objectives that the DSL could not only be an academic prototype, but we would also need to provide a language ecosystem with support tooling if it were to be used by data practitioners. Our goal was to have one major integrated development environment (IDE) that supported the language with syntax highlighting and completion hints, a design decision that influenced our technology choices as explained later in section 5.4.

5.3 Development Process

During development, we followed an open-source style development process using GitHub to coordinate activities.

After we gathered an increasing amount of feedback and knowledge about the language design approaches that worked, we synthesized them into a *Jayvee Manifesto*[3] that acts as a guideline to make self-directed decisions of each individual contributor. At the time of writing, the main design principles in this document (formulated as approaches we have come to value over others, inspired by the agile manifesto) are:

1. Describing a goal state over how to get there.

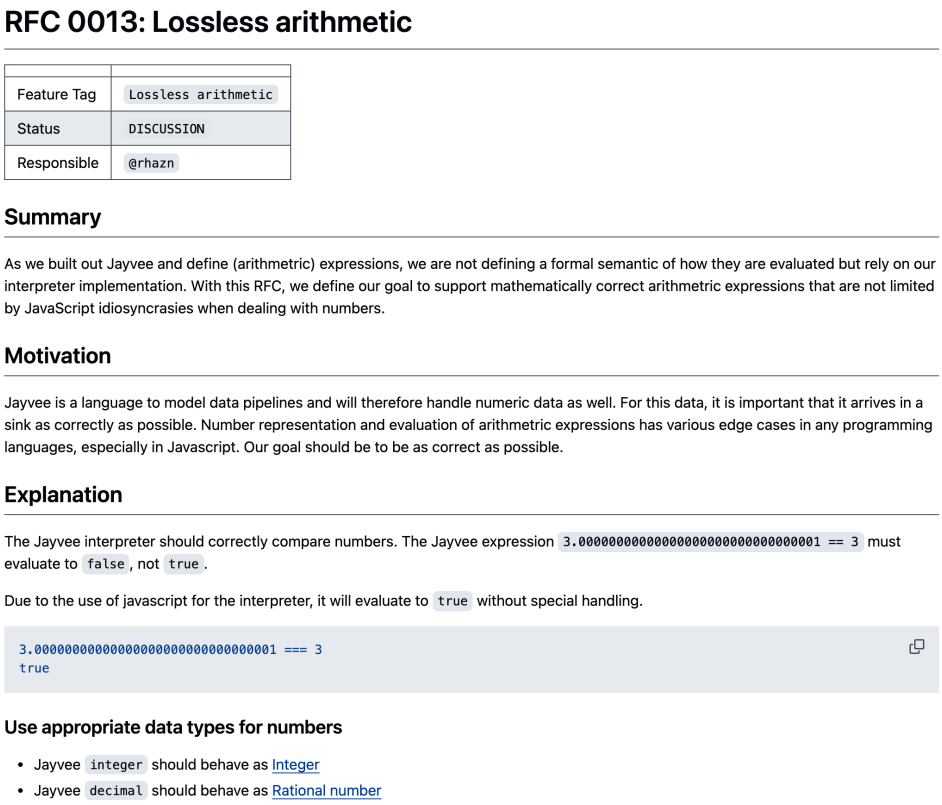
[3] *https://jvalue.github.io/jayvee/docs/0.5.0/dev/design-principles*

2. Explicit modeling over hidden magic.

3. Composition over inheritance.

4. Flat structures over deep nesting.

Long form discussions to evolve the language design were done using Request for Comments (RFC) documents[4]. During the RFC process, a proposed change is described in detail by a responsible developer using a Markdown document in the Jayvee repository. Figure 5.2 shows a part of an RFC document for reference.



RFC 0013: Lossless arithmetic

Feature Tag	`Lossless arithmetic`
Status	`DISCUSSION`
Responsible	`@rhazn`

Summary

As we built out Jayvee and define (arithmetic) expressions, we are not defining a formal semantic of how they are evaluated but rely on our interpreter implementation. With this RFC, we define our goal to support mathematically correct arithmetic expressions that are not limited by JavaScript idiosyncrasies when dealing with numbers.

Motivation

Jayvee is a language to model data pipelines and will therefore handle numeric data as well. For this data, it is important that it arrives in a sink as correctly as possible. Number representation and evaluation of arithmetic expressions has various edge cases in any programming languages, especially in Javascript. Our goal should be to be as correct as possible.

Explanation

The Jayvee interpreter should correctly compare numbers. The Jayvee expression `3.00000000000000000000000000001 == 3` must evaluate to `false`, not `true`.

Due to the use of javascript for the interpreter, it will evaluate to `true` without special handling.

```
3.00000000000000000000000000001 === 3
true
```

Use appropriate data types for numbers

- Jayvee `integer` should behave as Integer
- Jayvee `decimal` should behave as Rational number

Figure 5.2: Screenshot of an RFC in discussion to introduce mathematically correct arithmetic expressions in Jayvee.

The document has a flexible structure but generally outlines motivations, a suggested solution, and possible alternatives. After creating an RFC in a `DRAFT` state, the RFC enters `DISCUSSION`

[4]An example RFC discussing potential block instantiation syntax can be found at *https://github.com/jvalue/jayvee/blob/c4d1e0c731046 5b1f71621 2e1a314e52e946bf6d/rfc/0016-block-instantiation-syntax/0016-block-instantiation-syntax.md*

within the developer community. During the discussion, contributors give feedback and request changes based on the language design guidelines presented earlier. Finally, the RFC is either ACCEPTED or REJECTED. If the RFC is accepted, the responsible developer creates an issue to track the implementation of the RFC. While typically the developer who prepared the RFC also implements it, any participant can contribute to resolving the issue. Figure 5.3 shows an overview of the process.

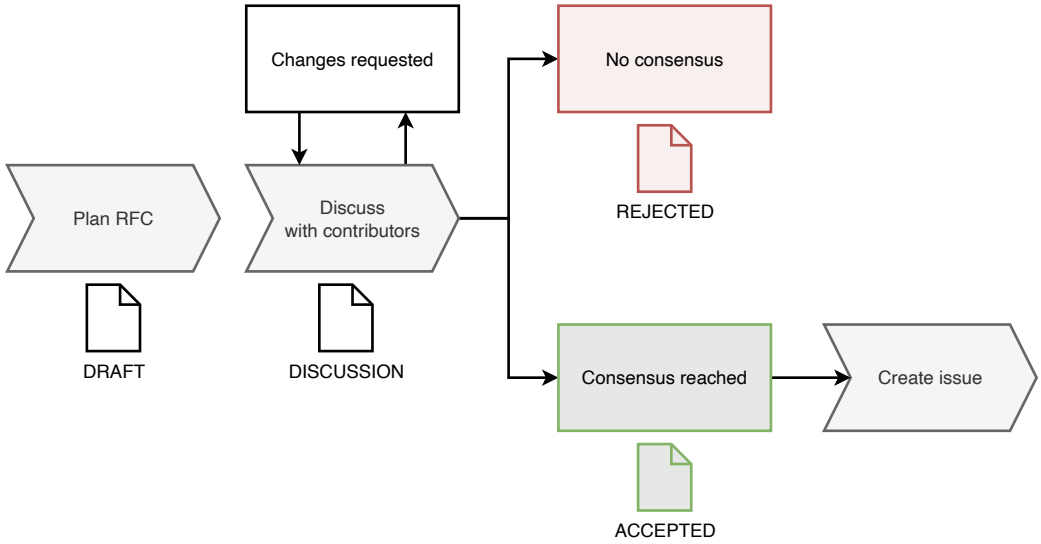


Figure 5.3: RFC process followed during the development of Jayvee.

In addition to this formal, asynchronous process, the core members of the development team also regularly met in person to discuss potential new features and a roadmap for Jayvee that was managed in an external project management tool.

5.4 Implementation Choices

We built Jayvee based on Eclipse Langium[5], a language engineering toolkit to create DSLs using TypeScript. The syntax for languages developed with Langium is defined using the Langium grammar language, which allows developers to describe context-free grammars similarly to extended Backus-Naur form notation. Listing 5.5 shows Jayvee language elements

[5] *https://langium.org*

defined in the Langium grammar language. In this excerpt, referenceable block types are defined as either composite or built-in block types (lines 1 - 2). Further, the syntax to implement these two types is defined in lines 4 - 7 and lines 9 - 19. Each uses a specific keyword (`builtin` and `composite` respectively), followed by the `blocktype` keyword, a name, and a list of possible properties.

Listing 5.5: Jayvee language elements, defined in the Langium grammar language.

```
1  ReferenceableBlockTypeDefinition:
2    BuiltinBlockTypeDefinition | CompositeBlockTypeDefinition;
3
4  BuiltinBlockTypeDefinition:
5    'builtin' 'blocktype' name=ID '{'
6      (inputs+=BlockTypeInput | outputs+=BlockTypeOutput | properties+=BlockTypeProperty)*
7  '}';
8
9  CompositeBlockTypeDefinition:
10   'composite' 'blocktype' name=ID '{'
11     (
12       inputs+=BlockTypeInput
13       | outputs+=BlockTypeOutput
14       | properties+=BlockTypeProperty
15       | blocks+=BlockDefinition
16       | pipes+=BlockTypePipeline
17       | transforms+=TransformDefinition
18     )*
19 '}';
```

With these definitions, Langium provides support for parsing language models into abstract syntax trees (ASTs). To interact with ASTs, Langium generates TypeScript classes for AST nodes and instantiates the concrete AST of a data pipeline model from them. We developed a language server using the language server protocol (LSP) based on the framework provided by Langium. To do so, we implemented validation rules for all elements of a pipeline model with helpful error messages and hints that can be used by IDEs to provide guidelines

for users. With these validation functions, we ensure that a Jayvee model is valid semantically. Listing 5.6 shows such a validation function that checks if a block type has more than one input and, if so, returns an error object with a message for the user on how to resolve the issue. An IDE such as VSCode uses this information to display semantic errors in a Jayvee model before it is executed.

Listing 5.6: A validation function for a block type definition in the Jayvee language server.

```
function checkNoMultipleInputs(
  blockType: ReferenceableBlockTypeDefinition,
  props: JayveeValidationProps,
): void {
  if (blockType.inputs === undefined) { return; }

  if (blockType.inputs.length > 1) {
    blockType.inputs.forEach((inputDefinition) => {
      props.validationContext.accept(
        'error',
        `Found more than one input definition in block type '${blockType.name}'`,
        { node: inputDefinition },
      );
    });
  }
}
```

Based on our goals, we chose Langium in order to provide a language ecosystem of tools to support users, such as at least one integration into a popular IDE. By implementing a language server based on the LSP, we could implement an open-source plugin for VSCode ourselves and enable other developers to write plugins for their favorite editors. In addition, being able to use TypeScript to interact with the AST and implement the LSP allowed us to make use of our previous knowledge and made language development accessible for outside contributors, especially students, who often have experience with web development. Additionally, we could provide easy-to-use web editors to work on Jayvee models based on the Monaco editor project.

The language semantics of Jayvee are defined by a reference implementation of an interpreter that traverses the instantiated AST and executes code depending on block type. The data that flows through an instantiated pipeline is then transferred from block to block and finally written into a sink. Figure 5.4 shows an overview of the technical implementation described here. For any Jayvee model, the Langium framework is used to perform lexical and syntax analysis, converting the input text into tokens and ensuring the model is syntactically correct before instantiating an AST from the input. Based on this AST, our Jayvee language server implementation can perform semantic analysis to verify that the model is semantically correct as well. In the final step, the Jayvee interpreter uses the AST to interpret the model and generates output in the form of data written into sinks.

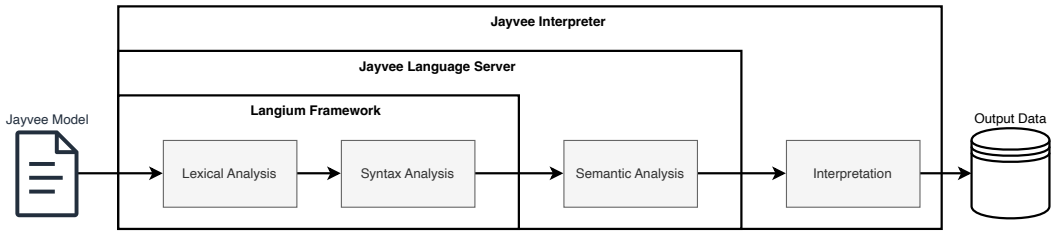


Figure 5.4: Overview of the phases and projects involved in interpreting a Jayvee model, adapted from the Jayvee developer documentation.

6

Demonstration

As explained in section 3.2, during demonstration, an artifact is used to solve exemplary instances of the problem under study to illustrate its use and show its feasibility. A thorough evaluation follows in chapter 7. The demonstration is less formal than the evaluation, and the main outcome is the knowledge that the artifact is a possible solution to the identified problem. Here we present a project with an industry partner in section 6.1 and how we involved Jayvee in teaching in section 6.2.

6.1 In Industry: Material Science Data with Springer Nature

One part of the demonstration was completed as part of the Software Campus collaboration with Springer Nature (see section 3.2). To demonstrate Jayvee with subject-matter experts in the material science domain, we explored a set of potentially valuable open science datasets provided by Springer Materials. The full demonstration, including example Jayvee models, can be found in a GitHub project[1].

[1] *https://github.com/jvalue/Exploring-Materials-Science-Data-Using-Jayvee*

The Jayvee interpreter creates data pipelines from these models that download and transform open datasets. These datasets are mainly based on data from scientific literature that was extracted automatically and needs to be cleaned up. Using Jayvee, we defined domain-specific value types such as digital object identifiers (DOIs), standardized DOI values, and introduced constraints on some columns to filter out invalid data. On every new release, the data pipelines get automatically executed using GitHub actions, and their output datasets are attached to the release. The cleaned datasets can be used in follow-up projects, such as the example material science data report notebook that is included in the demonstration.

We presented the results of the project at the Springer Nature Campus in Heidelberg. Afterward, we followed up with an online questionnaire according to the guidelines for a descriptive survey by Kitchenham and Pfleeger [31] (previously described in subsection 7.1.2) with our industry contact at Springer Materials. We asked about their agreement (on a 5-point Likert scale from *Strongly Agree* to *Strongly Disagree*) regarding the problems we identified (see also chapter 4) and free text feedback about the approach of developing a DSL and Jayvee itself. Our industry expert answered *Strongly Agree* to the challenges of a missing standard collaboration artifact and the need for specialized skills, but high barriers to entry. In response to DSLs themselves, they expressed having used a DSL before and "the idea of having a domain-specific language is a very good one", but cautioned "A challenge is to find the right balance between 'specialization' to a domain, and flexibility to transfer artifacts/code/workflows to other, related domains. Also, the domain-specific language needs to be integrated and used in the complete ecosystem of tools and data flows - otherwise, it may just end up adding complication." Jayvee itself was considered to "[...] look very promising".

6.2 In Teaching: The Methods of Advanced Data Engineering Module

In the summer semester 2023, we created a university module called *Methods of Advanced Data Engineering* (MADE) and have taught it for four semesters since. The goal of the module was to share knowledge about open data and advanced data engineering and software en-

gineering practices, such as automated testing, as they apply to data engineering. We mainly taught technology-agnostic concepts, but the language used in examples and the suggested prerequisite is Python. In addition to Python, we introduced how to read and write Jayvee code in two lectures. At the time of writing, 422 students, mostly from master's degrees in artificial intelligence and data science, have completed the module.

For the applied project work in MADE, students complete a self-directed data science report using open data, which describes their search for data, any required data engineering, and finally analyzes the data to answer a research question of the students' choice. While most students use Python for data engineering, like extracting and cleaning the data they are using, some have also created data pipelines in Jayvee. In addition to the main project, students were required to solve five data engineering exercises based on real-world open datasets during the semester. To pass these exercises, students had to implement a data pipeline that extracts the dataset, transforms it according to the exercise requirements, and writes the data to a sink. As an example, a slightly shortened exercise description is included here. In this exercise, the students used an open dataset about planted trees in Neuss. They were asked to:

- Define types of values and constraints for valid data
- Only include rows if stadtteil starts with Vogelsang
- id contains geopoints with the following pattern: {geo-coordinate 1}, {geo-coordinate 2}.
- A geo-coordinate is defined as {1-3 numbers}.{numbers}
- Drop the baumart_deutsch column and all rows with invalid values
- Assign fitting built-in SQLite types to all columns
- Write data into a SQLite database

During the first two semesters, participants in the MADE module were split into two groups and submitted exercises alternating in Python and Jayvee, demonstrating that Jayvee could be used to solve the same data engineering challenges as Python. In later semesters, only Jayvee was permitted to solve exercises. A sample solution using Jayvee to model a data pipeline for this is shown in Listing 6.1.

Listing 6.1: Sample solution for an exercise in the MADE module.

```
constraint DistrictRegex on text: value matches /Vogelsang[.]*/;
valuetype VogelsangDistrict oftype text { constraints: [ DistrictRegex ]; }

constraint GeoPointRegex on text: value matches /[0-9]{1,3}\.[0-9]+, [0-9]{1,3}\.[0-9]+/;
valuetype GeoPoint oftype text { constraints: [ GeoPointRegex ]; }

pipeline TreePipeline {
  Extractor
    ->TableParser
    ->Sink;

  block Extractor oftype CSVExtractor {
    url: "https://opendata.rhein-kreis-neuss.de/api/v2/catalog/datasets/stadt-neuss-herbstpflanzung
         -2023/exports/csv";
    delimiter: ';';
    enclosing: '"';
  }

  block TableParser oftype TableInterpreter {
    header: true;
    columns: [
      'lfd_nr' oftype integer,
      'stadtteil' oftype VogelsangDistrict,
      'standort' oftype text,
      'baumart_botanisch' oftype text,
      'id' oftype GeoPoint,
      'baumfamilie' oftype text,
    ];
  }

  block Sink oftype SQLiteLoader {
    table: "trees";
    file: "trees.sqlite";
  }
}
```

Participants in the MADE module create open-source projects. Because students are asked to fork the MADE template repository to use the provided structure, a list of most projects with exercise submissions is available on GitHub[2]. We consider the student population in MADE as a proxy for the target audience of Jayvee (discussed in more detail in the later evaluation, subsection 7.1.1). Their success in completing the data engineering exercises is a good demonstration of the accessibility of Jayvee for non-professional programmers, both for being able to learn the new language quickly and for using it to create working data pipelines.

[2] *https://github.com/jvalue/made-template/forks*

7

Evaluation

We approached the evaluation of Jayvee in multiple steps, where we combined qualitative and quantitative methods that are described in section 7.1. First, we validated that a DSL is a potential alternative to GPLs for building data pipelines to answer RQ2 (section 3.1). During the use in the context of the MADE module (see section 6.2), we gathered data about how the students solved their exercises using the DSL to verify that their performance is at a minimum comparable to using a GPL.

For the evaluation, we started with an initial exploratory phase, in which we gathered qualitative data about the effects of using Jayvee during data engineering. We did so to develop further, more focused hypothesis tests and, following the advice of Greenberg and Buxton [15], to not commit to a narrow, statistical evaluation of individual features too early. Because we worked iteratively, we could also learn from the insights for further revisions of the language before evaluating concrete features with controlled experiments. This work is described in section 7.2.

After generating concrete hypotheses to test, we conducted two controlled experiments

to evaluate individual design decisions. On a higher abstraction level, we considered an improved understanding of program structure by subject-matter experts as essential for using a DSL. Accordingly, we designed a mixed-method study including a controlled experiment to understand if and how the program understanding by subject-matter experts is influenced by the use of the DSL, presented in section 7.3. Next, we evaluated a specific syntax decision we made during design and implementation - the use of domain-specific cell selection syntax inspired by spreadsheet syntax - again with a controlled experiment in section 7.4.

This development of potential hypotheses using mixed-methods research, with a follow-up stream of controlled experiments with human participants, is the essence of the language development process with an iterative, empirical approach we developed. It is based on the test bed we have created with our custom DSL and the implementation of a web-based experiment tool that allows us to quickly develop new tasks and track automated measurements[1]. In future work, we plan to continue to use this process to evaluate more concrete features, such as the best way to express user-defined value types.

The combined insights from multiple evaluation studies of individual design decisions on different abstraction levels allowed us to develop a deeper understanding of their effects, answering RQ3 about important considerations for a DSL to create data pipelines by subject-matter experts (section 3.1).

An overview of the research approach is shown in Figure 7.1.

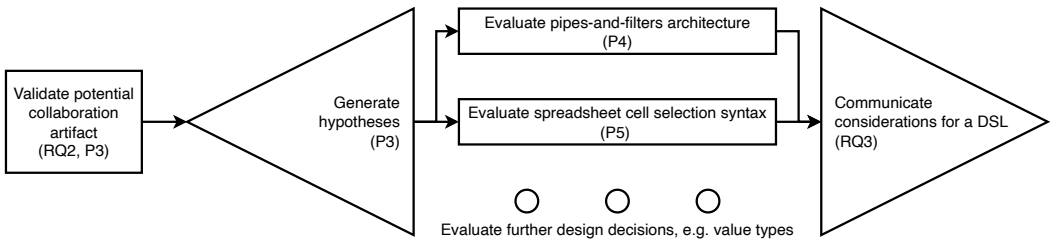


Figure 7.1: Research approach during evaluation and how it relates to research questions (RQ, section 3.1) and publications (P, Table 3.1).

[1]The experiment tool's source code is available as part of the data release for Heltweg et al. [18] at *https://doi.org/10.5281/zenodo.15543965*

7.1 Research Methods

During the evaluation, we used a combination of qualitative and quantitative research methods. We conducted surveys according to Kitchenham and Pfleeger [31] to get qualitative feedback from users and used thematic analysis according to Braun and Clarke [4] to analyze it and identify common themes. Additionally, we ran controlled experiments according to Ko et al. [33] to gather qualitative data about participants' performance and test concrete hypotheses.

7.1.1 Population Description

For the evaluation studies, we sampled participants from the MADE module on advanced methods of data engineering, during which we also demonstrated the use of Jayvee for homework exercises, described earlier in section 6.2. The students mainly pursue master's degree programs in artificial intelligence and data science, with some related disciplines like information systems and computer science. They generally have some previous experience in using Python for small data science applications, but are not professional system programmers due to the structure of their degree programs.

We consider these students a good proxy for data practitioners who have some experience writing scripts to work with open data, but are experts in their subject area and not software engineers. As described in chapter 4, open data practitioners come from a wide range of backgrounds and include subject-matter experts without a technical background. Similarly, the students might have some previous experience in programming and data engineering, but are not professionals in either [11].

However, while we expect the evaluation to generalize well to this limited population, we caution against assuming that professional programmers and other participants with a strong technical background face similar challenges. In future work, we want to replicate the existing experiment designs with professionals working with open data in industry projects to strengthen the results.

7.1.2 Descriptive Survey

Surveys, according to Kitchenham and Pfleeger [31], are a research method to gather quantitative data about subjective opinions of participants, or objective data such as demographics, typically using an online questionnaire. For this, Kitchenham and Pfleeger describe six steps: Setting objectives, survey design, developing and evaluating the questionnaire, obtaining the data, and analyzing the data.

Objectives are typically descriptive, such as a frequency or the severity of a characteristic in a population, but survey goals can also include identifying factors that influence a condition. In either case, before designing a survey, a research question and a way to answer it using measurements must be defined. During survey design, the survey itself is planned. Common designs include cross-sectional surveys (across a population at one fixed point in time) and longitudinal (repeated surveys at different points in time) studies. In addition, the method of administration must be clarified. Nowadays, most surveys are run using a self-administered online questionnaire.

Developing the questionnaire starts by first searching existing literature for relevant work and previously validated measurement instruments and questions. Afterward, questions are defined, which can be either open (for respondents to answer freely) or require respondents to select answers from a list. Answers can be numerical values, categories, binary (Yes/No questions), or ordinal scales. A common form of ordinal scales are Likert scales, on which respondents select from an ordered list of choices, often labeled with both a number and description. Likert scales can measure agreement with a statement (e.g., from Disagree to Agree), frequency of occurrence (Never to Always), or evaluate a concept (Terrible to Excellent). Questions should be precise and unambiguous, and the target population must be able to understand them correctly. To this end, questions should be as simple as possible and accompanied by appropriate instructions.

Once a first version of the questionnaire is developed, it can be evaluated using focus groups or pilot studies in which the questionnaire is used with a smaller sample to test it for being un-

derstandable and of appropriate length. This process can be iterative, where each pilot study leads to a changed questionnaire that is then evaluated again. Finally, the survey instrument must be documented, e.g., by writing a questionnaire specification that includes the objectives of the study, rationale for each element of the questionnaire, and how it was evaluated.

Obtaining the data is the activity of actually administering the questionnaire. Because it is typically impossible to survey the whole target population, a subset sample has to be selected. The main concerns for selecting a sample are the avoidance of bias, the sample being appropriate to the target population, and cost-effectiveness to be able to actually administer the survey. A variety of sampling strategies exist, including probabilistic sampling methods (e.g., a simple random sample), cluster-based sampling (individuals of defined groups are sampled), or non-probabilistic sampling. Non-probabilistic sampling, such as convenience sampling, runs the risk of being biased or not representative of the target population, but is reasonable to use if the target population is hard to define or not easily available. After defining a sample, the survey instrument is administered. During administration, response rates should be tracked and reported because a low response rate can introduce bias to the results (e.g., is the pool of respondents still representative?).

Once the data is obtained, it has to be analyzed. Even though this activity is the final step of a survey, it should be planned before the data is actually collected. First, the data is validated, and inconsistent or incomplete questionnaires are handled according to a predefined strategy. With the validated data, statistical methods such as calculating measures of central tendency or statistical tests can be used to describe the content and test hypotheses.

7.1.3 Thematic Analysis

Thematic analysis, according to Braun and Clarke [4], is a research method to analyze qualitative data for meaning across the entire dataset. It is a flexible and conceptually simple method that is accessible for researchers with varying degrees of familiarity with qualitative methods. The core concepts of thematic analysis are codes and themes. Codes are annotations in the

form of a short label on a subsection of the source data. Codes can be descriptive and directly based on the content expressed in the data, or interpretative, meaning they capture an underlying meaning in the data based on the researcher's understanding. Due to the connection between a code and the annotated data, every code consists of a label and excerpts from the data that illustrate its meaning. Themes are a collection of codes that describe a pattern or a concept in the dataset that is relevant to the research question. They consist of a descriptive title and, ideally, a so-called *thick* description, meaning explicitly chosen citations from the source data that express the theme well, and an additional analysis by the researcher that summarizes how and why the theme is relevant to the research question is presented.

Using these building blocks, thematic analysis can be approached in an inductive or deductive manner. When working in an inductive way, researchers create codes and themes bottom-up from the content of the data itself. If the researchers have a preconceived theory or topics on which they base their interpretation of the data, they work in a top-down manner and follow a deductive approach.

The thematic analysis process, as described by Braun and Clarke [4], consists of six steps. First, the researchers familiarize themselves with the data by actively engaging with it and taking personal notes. If the qualitative data exists in the form of interviews, the act of transcribing can be a form of familiarization with the dataset. At the end of this step, the researchers should have a good overview of the data. Second, initial codes are generated. During this activity, coding should be inclusive and, if in doubt, codes should rather be created than not. Each data item should be completely coded before moving on to the next, but codes can be continuously modified based on new data, and old data can be revised to apply new or modified codes. At the end of this step, the dataset should be fully annotated with many codes that capture the diversity of concepts in the data as they relate to the research question.

Next, the initial codes are used to search for themes. In this step, the researchers actively build themes from the codes, for example, by combining multiple codes that share a common feature. In this process, the researchers aim to describe the overall insights that can be gathered

from the data with themes and their relationships to each other. There is no clear guideline on how many themes should be created, but themes should be relevant to the research question and not simply a summary of every topic that exists in the data. After searching for themes, these are reviewed for quality and coherence. To do so, the candidate themes from the initial search are first checked against the codes. If the theme has enough support to stand on its own (and should not just be a code), is of high quality with clear boundaries and a singular focus, it is kept. In a final review, the theme is then checked against the full dataset again. Themes that capture a meaningful aspect of the dataset are retained.

In the final data analysis step, the remaining themes are clearly defined and named. During this activity, a descriptive name should be assigned to every theme and the aforementioned thick description, consisting of exemplary citations and an analysis by the researcher. The themes retained at this point should have a clear focus and boundaries, be related to each other, and address the research question. They can be largely descriptive, based only on direct excerpts, or interpretative and include concepts derived by the researcher. In both cases, the combination of themes and the analysis they contain should build on the data and add additional insights towards the research question instead of being only a summary. After all themes are collected, the last step of thematic analysis is to produce a report, typically as part of an academic publication. The report puts the themes in the larger context of a research study and answers the research questions. For this, the themes are actively organized in a way that makes the insights found in the data most clear.

7.1.4 Controlled Experiment

A controlled experiment, according to Ko et al. [33], is a comparative method to test clearly defined hypotheses based on quantitative measurements. The authors provide guidelines for controlled experiments with human participants to evaluate new software engineering tools. Experiments determine if the application of a treatment (a change in an independent variable) leads to a change in a dependent variable. Common to any experimental research design is

the core tradeoff of controlling as much of the environment as possible versus providing a realistic environment. Controlled experiments err on the side of controlling as much of the environment as possible, and therefore, they provide a way to rigorously test hypotheses, but are more challenging to generalize to real-world contexts.

Even before running a concrete experiment procedure, the experiment should be defined, including clear goals, formal hypotheses and variables, as well as the experiment context and its participants [63]. Depending on these choices, a fitting experiment design, such as a crossover design [58], will determine how participant groups are formed and how treatments are applied.

After planning, Ko et al. [33] describe the controlled experiment process in eight steps, starting with gathering experiment participants by recruiting them, testing them against inclusion criteria, and then asking for their informed consent. The experiment procedure itself consists of gathering demographic data, assigning participants to a group, training them, having participants complete tasks, and finally debriefing them.

Recruiting participants can be problematic for experiments in software engineering because software engineers in industry are relatively well compensated, and it is hard to convince them to spend time participating in an experiment. As an alternative, experiments with student participants can be a good way to test initial hypotheses, as long as they are chosen with a clearly defined target population in mind [11]. In any case, participants should be filtered by inclusion criteria, such as programming experience or previous work in related technologies.

It is imperative to follow ethical research practices when working with humans. While formal ethics approval is often not required in software engineering experiments due to their low risks for harm and their limited focus on sensitive populations or topics, every participant must be able to give informed consent. Typically, this is accomplished by providing a handout that details the study and any risks, how data will be managed, and what benefits are expected from the results. Additionally, researchers must create opportunities for participants to ask any questions they might have about the experiment and to withdraw their consent without

any negative effects at any time.

After participants are recruited and have given their consent, the experiment procedure is executed. The exact experiment design will determine how many and which different treatments and groups exist. First, participants are assigned to treatments, most often by simple random assignment. Afterward, the participants must be trained so they have all the knowledge required to succeed in their tasks. Either before or after the experiment procedure, additional demographic data can be gathered. Next, participants solve tasks in controlled conditions using the treatment that is assigned to them. The environment can be more or less controlled (e.g., in a lab setting versus at home), with researchers typically favoring control.

The tasks themselves can be designed from scratch or be realistic examples from real-world problems, such as existing open datasets. It is important to prepare tasks carefully so they are neither too hard nor too easy and can be completed by participants in a realistic timeframe. During task attempts, outcome variables are measured, either by the researchers themselves or automatically using an automated experiment tool. Typically, outcome variables can be categorized into different *success on task* and *time on task* measurements. Because these categories are tradeoffs (the more time participants spend on a task, the better it can be solved), researchers must decide on how to guide participants.

Finally, after all participants have completed the tasks, they are debriefed with additional information about the study, such as goals, correct solutions, and the chance to use the new tool as well.

Since experiments are complex to set up and hard to execute, Ko et al. [33] suggest iteratively developing the experiment tasks and procedures using pilot studies with other researchers or with smaller subsets of the population under study. This approach also allows for designing better experiment instructions for participants and making sure tasks are not too difficult to complete. Reporting experiments typically happens as part of a publication. Various reporting templates, such as by Jedlitschka and Pfahl [25], exist to increase comparability and make sure researchers include all important information.

7.2 Domain-Specific Languages as a Viable Basis for a Collaboration Artifact

The focus of our first exploratory evaluation study was to verify if a domain-specific language can be a viable basis for a collaboration artifact for subject-matter experts who contribute to collaborative data engineering projects. Additionally, we were interested in the effects that users of a DSL are experiencing when creating data pipelines. Our goal was to adapt our language implementation and generate future hypotheses to test, based on these insights.

The following sections are a summary of the work published in Heltweg et al. [20]; more details are available in the original paper (see Appendix C).

7.2.1 Study Design

We chose a mixed-methods research design that allowed us to mitigate the weaknesses of the individual methods and strengthen our results with insights from multiple data sources [27]. The research design had three phases to gather qualitative and quantitative data before analyzing the results.

First, we collected quantitative data in a cross-sectional, descriptive survey according to Kitchenham and Pfleeger [31] (see subsection 7.1.2). We used convenience sampling with MADE students (see subsection 7.1.1) and gathered data on their previous programming experience before randomly assigning them to two groups. Each week, one group solved the data engineering exercises using Python and Pandas while the other used Jayvee. For the next exercise, the language assignments were swapped. After each exercise, we asked students to voluntarily give us feedback using an online survey. The survey questionnaire asked how much time students spent on the exercise, their impression of difficulty, and the quality of the result (both on a 5-point Likert scale). In preparation for follow-up interviews, we also included free-text questions about problems the participants encountered and suggestions for improvements. In this way, students could provide further feedback about the language and their experience using it.

Second, we followed up with a qualitative survey (see subsection 4.1.2) according to Jansen [24] after the semester had completed. Based on the data from the descriptive survey, we developed an interview guide according to Kallio et al. [28] and held semi-structured interviews about difficulty, quality, and experienced challenges with volunteers.

Lastly, we analyzed the resulting quantitative data with statistical methods (using descriptive statistics and hypothesis tests) and used thematic analysis according to Braun and Clarke [4] (see subsection 7.1.3) to analyze the qualitative data for potential reasons for the results and summarize challenges the participants experienced. Because we had no preconceived theory of the effects of using a DSL for data engineering, we chose an inductive approach to coding, annotating codes according to what we found in the data, and then iteratively combining them into major themes.

An overview of the research design is shown in Figure 7.2.

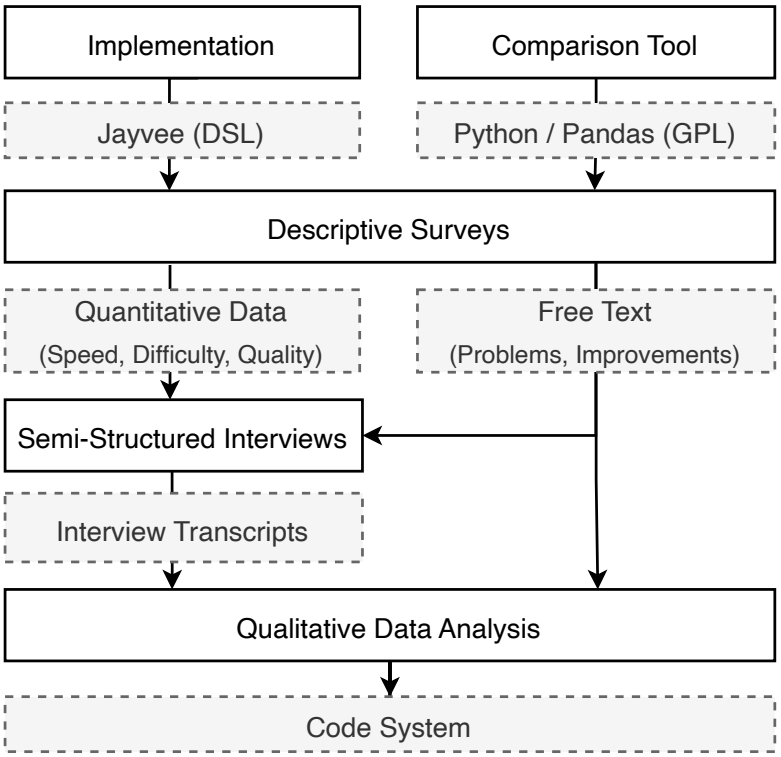


Figure 7.2: Study design as performed for Heltweg et al. [20].

7.2.2 Improved Performance Without Previous Experience

We analyzed the results of the descriptive survey with appropriate hypothesis tests to compare the participants' previous experience in Python and Jayvee and to evaluate if their performance on individual exercises had improved with regard to perceived quality, speed, or difficulty. As expected, participating students had significantly less experience in Jayvee than in Python on a scale from 1 *very inexperienced* to 5 *very experienced* ($n = 223, Mdn_{Python} = 4$, $Mdn_{Jayvee} = 1, p \leq .001$). For most exercises, students performed similarly when using Jayvee or Python. However, in some exercises, students performed significantly better using Jayvee. Participants solved exercise 2 faster ($n = 52, p = .042$), considered exercise 4 easier ($n = 35$, $p = .040$), and their solution to exercise 1 of higher quality ($n = 95, p = .021$). For a more detailed description of the exercises themselves, please refer to Appendix C.

In their interviews, participants rarely mentioned any impressions regarding implementation speed, but considered Jayvee faster for smaller problems and generally slightly preferred the terse syntax of Python for larger ones. Jayvee was described as easier because it was quick to learn and had a much more limited feature set (and no libraries) in comparison to Python. Additionally, sometimes the automated, hidden logic employed by Python and Pandas meant students could not understand their code clearly and had to double-check the output of their pipeline.

With regard to the quality of the result, participants especially highlighted the stronger enforced structure that Jayvee provided. This consistent framework was able to guide development and remind students about the steps they needed to perform. As an additional benefit, the data pipeline code was always broken up into smaller, coherent pieces that were easier to understand and reuse. The results of the qualitative data analysis also reinforced our impression that a language ecosystem including good documentation, IDE integration, and a debugger is essential for a DSL to succeed with real users.

7.2.3 Easier Collaboration

For completeness, an overview of the full code system of the thematic analysis is shown in Figure 7.3. Here, we highlight a subset of important results regarding the research questions of this thesis. For a full discussion of the thematic analysis, see the complete publication in Appendix C.

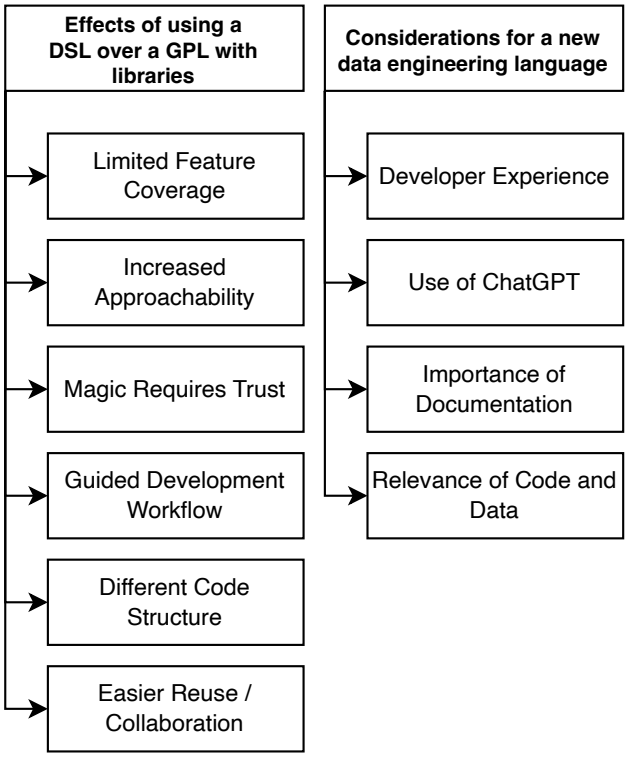


Figure 7.3: Effects of using a pipes-and-filters based DSL on data engineering (adapted from [20]).

Even though collaboration was not a topic we specifically asked about, some participants shared that they felt that the block-based structure of Jayvee made it easier to reuse previously written code for future pipelines. Furthermore, they mentioned that this easier reuse would also enable collaborative work. The main reason for this impression was the consistent bundling of all code related to a step in the pipeline in a block. While this structure can be

used in Python as well (for example, by using classes or functions), often smaller scripts are written in an imperative manner and lack a structure that makes reuse easy.

In addition, participants described user-defined value types as easier to understand and reuse than writing custom callbacks using if statements in Python code. Lastly, Jayvee is scoped only to data engineering and does not provide an additional library system, which means that only one way to achieve a goal is available. The extensive availability of libraries (such as Pandas) and their different approaches to problem-solving were described as hindrances to understanding code written by other participants in Python.

7.2.4 Strongly Enforced Code Structure Guides Development

When talking about why Jayvee was easy to understand and write, a major theme mentioned by interviewees was the strongly enforced code structure in comparison to the flexibility of Python. While writing code, students experienced this structure as a guideline or reminder for steps they still needed to complete before a pipeline was done.

The largest effect was described in regard to the readability of source code, with the structured overview by wiring up blocks using the pipe syntax mentioned as especially important. The block structure itself leads to smaller units of connected code that are easier to understand in isolation than script-style programming. In combination, the enforced structure of Jayvee seemed to allow participants to understand source code better and consider their results of higher quality.

7.2.5 Approachability and Relevant Experience Outside of Software Engineering

Students mentioned various reasons for Jayvee being easy to approach and learn. Since the design of Jayvee was directly inspired by the mental model of data pipelines as connected processing steps (and the visual representation as a DAG, see also section 5.2), some participants described that they could reuse experience from data pipeline modeling tools they used before.

Additionally, multiple participants mentioned that they learned data engineering using spreadsheet tools and still use them when exploring and cleaning datasets. This matches our experience of how data practitioners work with open data (see section 4.2). During the design of Jayvee, we decided to use domain-specific syntax, inspired by spreadsheet tools, to select cells in 2D data structures. This choice was highlighted by participants as reminding them of spreadsheet software and making it easier to select cells in Jayvee, rather than having to program in a GPL.

7.2.6 Conclusion and Future Research

In conclusion, we documented that a DSL such as Jayvee is a potential choice for data engineering by non-professional programmers, with the quantitative results showing that it can be learned quickly and is usable with slight improvements to solve small data pipeline creation tasks. The positive feedback on the effects of collaboration is an indicator that source code written in Jayvee can be a viable foundation for a collaboration artifact in data engineering (RQ2, section 3.1).

In future research, we want to follow up with more focused studies on individual effects.

First, Jayvee code was described as easier to understand, with major effects from the stricter enforcement of code structure, and less hidden logic. However, of course, tradeoffs exist, such as less flexible and more verbose code. To determine if these tradeoffs are worthwhile, we designed a mixed-method study to evaluate the effect size and the ways that a DSL can influence program structure comprehension, presented in section 7.3.

Second, domain-specific syntax was mentioned in the interviews as a reason for the approachability of Jayvee with previous experiences outside of software engineering. Especially the spreadsheet syntax used to select cells from 2D data in Jayvee is relevant, because data practitioners also regularly use spreadsheet tools to complete data engineering tasks. We planned a focused, controlled experiment to test the hypothesis that spreadsheet syntax improves task performance for data practitioners compared to numeric indexing often found in GPLs and

share the main results in section 7.4.

7.3 Program Structure Understanding

A main effect of using Jayvee to complete data engineering tasks, as described by interviewees, is the lower difficulty of understanding the source code. Moreover, program understanding is a core activity during programming [51, 65], and to collaborate with others, contributors must at the very least correctly understand the shared collaboration artifact. For this reason, we first designed a mixed-methods study to test our hypothesis that the use of Jayvee has an effect on bottom-up program structure comprehension and to describe in more detail any reasons for the effects. Here, we present relevant excerpts from the work published in Heltweg et al. [19], please see Appendix D for the complete manuscript.

7.3.1 Study Design

We chose a mixed-method design to combine the hypothesis tests with a search for potential explanations for the results. To start, we performed a controlled experiment according to Ko et al. [33] (see subsection 7.1.4). As discussed in chapter 4, we consider the design and evaluation of a new DSL from the viewpoint of empirical software engineering and were therefore interested in gathering data from users instead of making purely technical comparisons.

We chose to test if the use of Jayvee had an influence on how *fast* or *correct* non-professional programmers understand data pipelines compared to Python/Pandas and if the perceived *difficulty* of understanding the source code changed with both treatments.

Using the controlled experiment, we gathered two performance measurements, time on task and correctness, which are common standards in software engineering experiments [64]. For participants, we relied on a convenience sample of largely artificial intelligence and data science master's students from the MADE module as proxies for data practitioners (see subsection 7.1.1).

We chose a factorial crossover design in which every participant is assigned to each treat-

ment (within-subjects) [58]. To reduce carryover and familiarization effects, we created two sequences with a reversed order of treatment assignment to tasks and randomly assigned participants to sequences.

Tasks were designed based on real-life open datasets. We implemented comparable data pipelines to access them in Python/Pandas and Jayvee, and used pilot studies to refine the tasks themselves and their description. Participants used a web-based experiment tool developed by us to read the source code and to recreate the data pipeline using a drag-and-drop interface of possible steps. In addition to allowing us to provide a controlled environment for the participants, the tool also took automated measurements of time and correctness to ensure the variables were accurately tracked. An example screenshot from a program understanding task displayed in the experiment tool web interface is shown in Figure 7.4.

After the experiment, we asked participants to complete a descriptive survey according to Kitchenham and Pfleeger [31] (see subsection 7.1.2). Using this web-based survey, we gathered responses on the perceived level of difficulty (rated on a 5-point Likert scale) to complete the gathering of quantitative data for further hypothesis tests. In the same questionnaire, we included free-form questions about what makes data pipelines written in Jayvee/Python hard or easy to understand and what differences they experienced between the languages. The responses to these questions created the qualitative dataset we analyzed to explore potential reasons for the results of the hypothesis tests.

For hypothesis tests, we chose appropriate statistical methods (the Wilcoxon signed-rank test [60]) for the paired data from a crossover experiment design according to Wohlin et al. [63].

We analyzed the qualitative data from the survey using thematic analysis according to Braun and Clarke [4] (presented earlier in subsection 7.1.3). To do so, we chose an inductive approach with the goal of capturing the diversity of effects that had an influence on program understanding. During the process of coding and creating themes, we kept track of changes to our codebook and judged the maturity of our theory based on theoretical saturation [3].

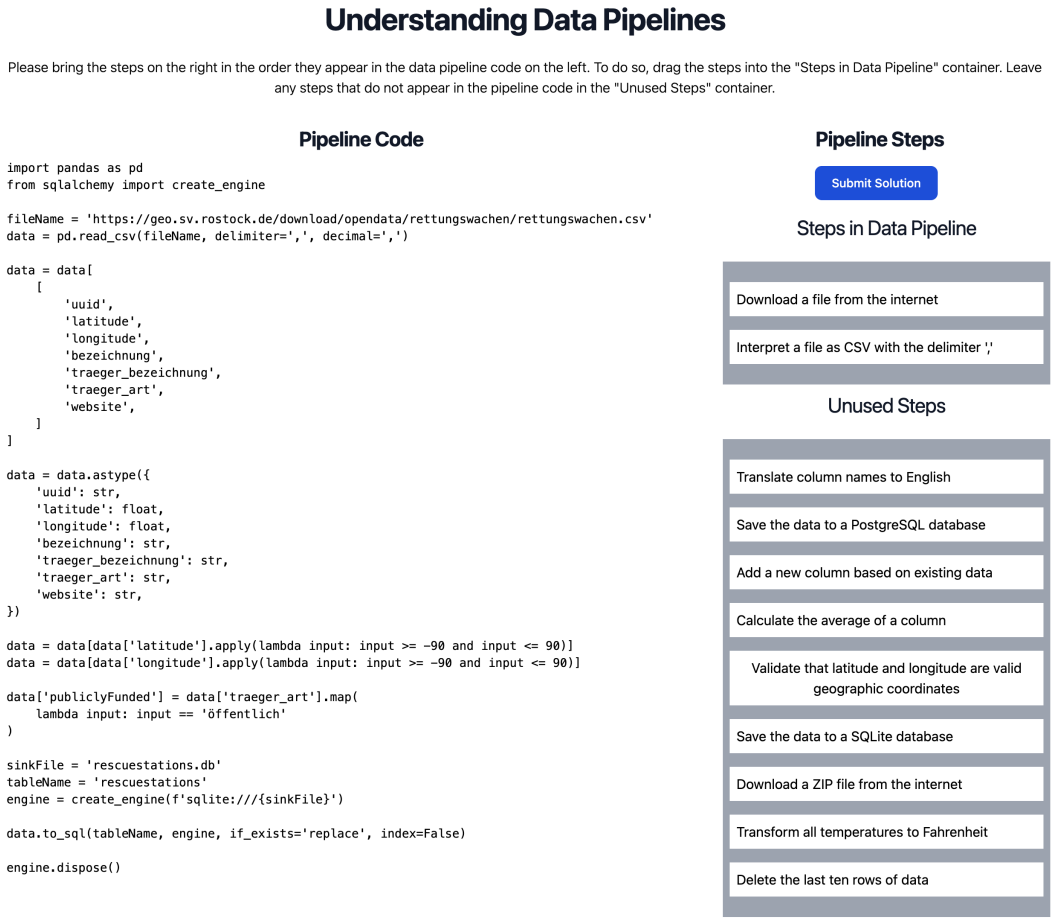


Figure 7.4: Screenshot of a program understanding task as it was displayed in the web-based experiment tool.

7.3.2 Improved Correctness but Not Faster

Participants could understand data pipelines expressed in Jayvee significantly more correctly than those expressed in Python/Pandas ($n = 57, p = .002$). The kernel density plot in Figure 7.5 shows the distributions of the data.

In contrast, there were no significant differences for time on task ($n = 57, p = .546$) or on perceived difficulty ($n = 56, p = .153$).

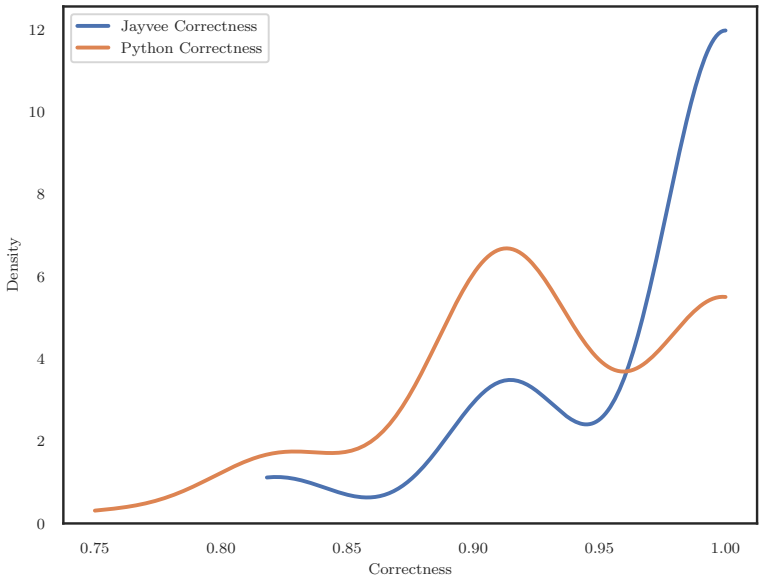


Figure 7.5: Kernel-density plot comparing correctness of task solution using Jayvee and Python/Pandas (adapted from [19]).

7.3.3 Code Structure and Language Elements

In addition to the hypothesis test results, we present in the following sections a selection of important insights from the thematic analysis. The complete codebook is shown in Figure 7.6 and is described in the manuscript in Appendix D.

In the context of direct effects of the language used (in contrast to human factors), the experiment participants highlighted the code structure as a positive influence on data pipeline understanding. Especially the pipeline overview, provided by wiring up the data pipeline graph using the pipe syntax, in a separate location of the source code, was described as an important difference to Python/Pandas. On the one hand, the separation allowed participants to ignore unnecessary code when reading the pipeline for the first time and contributed to a better overview. On the other hand, some participants described that the additional navigation required to jump from the pipeline overview into code details costs time.

While the pipeline overview was overwhelmingly identified as helpful, pipeline steps themselves must be correctly scoped. Participants were confused by steps like interpreting a downloaded file first as text and then as CSV, because they did not consider these low-level steps in

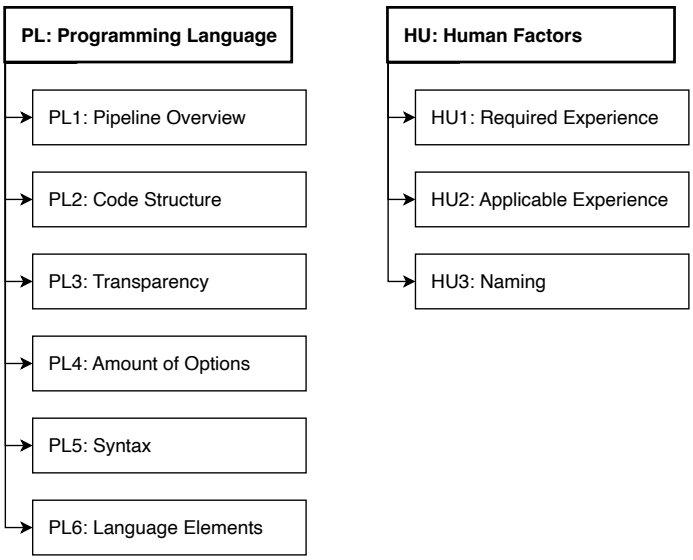


Figure 7.6: Reasons for differences in program structure understanding between Jayvee and Python/Pandas (adapted from [19]).

their own mental model of a data pipeline.

Regarding language elements, the participants considered domain-specific elements, such as blocks, as intuitive and described them as having a positive effect on understanding. In contrast, unusual language elements that are not a direct match to the application domain, such as Jayvee's value types based on constraints, confused participants. Similarly, participants identified advanced programming concepts, such as lambdas or list comprehension in Python source code, as reasons for reduced understanding. Very likely, this is especially true for data practitioners without a strong system programming background, but less of a challenge for more experienced programmers.

We observed that these language elements also had an effect on the experience needed to understand the data pipeline models. When represented in Python/Pandas, the source code is structured using programming concepts such as functions or library imports that require previous knowledge of software engineering. In comparison, participants mentioned that they felt like they could draw on other experiences, such as previous data engineering work, when reading Jayvee code.

Additionally, more flexibility and multiple approaches to solving a problem due to libraries and a more complex syntax meant that Python code was harder to understand for participants. They also mentioned that, while it is possible to structure Python code well, more experience would be required to follow best practices without a strongly enforced structure by the language itself. Some participants mentioned that, because Jayvee pipelines look very similar to each other due to this enforced structure, reading or implementing multiple pipelines provides more opportunities for learning effects. This is different in Python, where the added flexibility means that two pipelines by different developers do not necessarily follow similar patterns, and therefore learning effects are reduced.

7.3.4 The Right Level of Abstraction Is a Challenge

Participants further described the alignment of their mental model of data pipelines with the elements they found in the data pipeline model as improving their understanding of the source code. This effect existed even without a large amount of previous programming experience, but was based on an understanding of the underlying domain of data pipelines.

A repeating pattern in participants' responses was how domain-specific concepts they expected were intuitive and supported understanding, while concepts they did not consider were confusing. This means that finding the right level of abstraction based on the mental model of a data pipeline is a core challenge for designing a good DSL. One example for this effect is the previously mentioned pipeline overview, which was highlighted as a major positive influence on understanding, except when participants were confused by individual steps that were too low level to make sense to them (such as downloading a file as binary data, then interpreting it as text and CSV).

Importantly, this mental model depends on the users who work with the DSL. A language that aims to support data practitioners, such as subject-matter experts, must consider a higher level of abstraction than a language that is used by system programmers in regard to programming. However, it could instead include domain concepts that data practitioners are familiar

with from their work with data pipelines.

7.3.5 Conclusion

To summarize, we show that a DSL such as Jayvee can improve correctness for bottom-up program understanding of data pipeline models by data practitioners, such as subject-matter experts. Because program understanding is a central programming activity and a prerequisite for contributing to a shared collaboration artifact, a more correct understanding enables collaboration and can lead to fewer bugs and better downstream data.

In regard to RQ3 (see section 3.1 for details) about important considerations for DSLs for data engineering by subject-matter experts, our results highlight that finding the correct abstraction level for data practitioners is a major challenge. Designers of DSLs will need to invest in clearly defining their target users and understanding their mental models of the domain. In the domain of creating data pipelines, experiment participants seem to consider the blocks and pipes structure of Jayvee intuitive, but struggle with value types based on constraints and block types that are too low-level.

Lastly, the choice of language elements has a large effect on the usability of a DSL as well. In our experiment, GPL elements such as functions or complex programming concepts such as lambdas were considered negative, while domain-specific elements such as blocks had a positive influence. In the follow-up experiment (section 7.4), our goal was to build on this insight and validate whether domain-specific syntax, such as spreadsheet syntax to select a subset of cells in 2D data, also had positive effects on usability.

7.4 Cell Selection Syntax

During the problem identification and in our previous evaluation studies, we learned that data practitioners regularly use spreadsheet software for data engineering. We have also shown the importance of aligning language elements with the mental models of their users. To evaluate if using domain-specific syntax has a positive effect as well, we design a focused evaluation of

the design decision implemented in Jayvee to use spreadsheet syntax in order to select cells in 2D data.

The work presented here is a summary of relevant points from a manuscript published as Heltweg et al. [18]. Additional descriptions and results can be found in the original publication (see Appendix E).

7.4.1 Study Design

We designed and conducted a controlled experiment according to Ko et al. [33] (previously discussed in subsection 7.1.4). The experiment was again performed with students of the MADE module (see subsection 7.1.1), and we invited participants mainly from master's degrees in artificial intelligence and data science as proxies for data practitioners without a professional software engineering background.

Our goal was to evaluate if domain-specific syntax, represented by spreadsheet syntax such as `A5:B10`, is better for selecting cells in 2D data compared to the common numerical syntax found in Python/Pandas (based on iloc, e.g. `df.iloc[2:6, 1:4]`).

We decided on designing two sets of tasks, based on real-life open datasets, to measure task performance for program understanding on the one hand and for code creation on the other hand. The tasks involved participants reading cell selection code, marking the selected cells in a visual representation of the data, and seeing a visual representation and writing cell selection code to reproduce it. For both dimensions, we measured time spent on the task and correctness of the solution. Based on this data, we tested multiple hypotheses: Does the use of spreadsheet syntax have any influence on *speed or correctness* for either *program comprehension or code creation* by data practitioners compared to numerical selection syntax?

We adapted and reused our web-based experiment tool (an example task is displayed in Figure 7.7) for measurements and invited participants to a university-provided lab to offer a controlled environment. As an experiment design, we chose the crossover design [58] with participants assigned randomly to two sequences and completing multiple tasks with alternat-

ing treatments.

To analyze the data, we used Wilcoxon signed-rank tests [60] and explored the data distributions using kernel density plots.

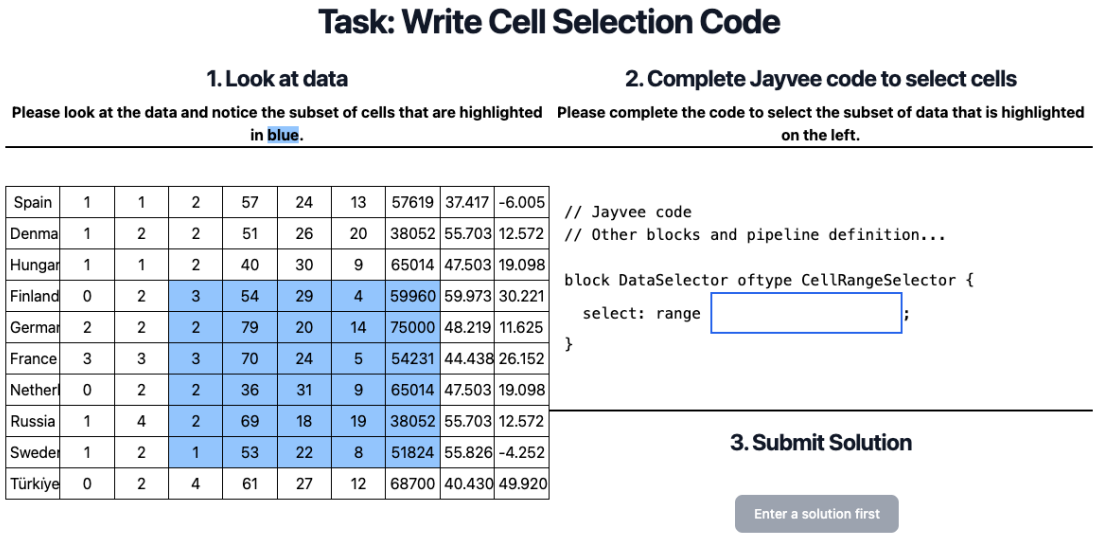


Figure 7.7: Screenshot of a code creation task as it was displayed in the web-based experiment tool.

7.4.2 Improved Correctness

Correctness of task solutions was significantly different for both program comprehension ($n = 83, p = .019$) and code creation ($n = 93, p \leq .001$). In both cases, the solutions based on spreadsheet syntax showed improved correctness compared to the numerical cell selection syntax.

For program comprehension tasks, participants did not submit solutions at significantly different speeds. This could indicate that both syntax approaches appear to be similarly easy to read and understand, while the numerical indexing actually introduces subtle errors, mostly with off-by-one errors.

The improved correctness for program comprehension using Jayvee mirrors the previous results from section 7.3 in a new, more focused context. The extension of improved correctness to code creation using spreadsheet syntax was a new insight from this study.

7.4.3 Code Creation Is Faster Using Spreadsheet Syntax

In contrast to program comprehension, time on task for code creation was significantly different between both treatments ($n = 84, p \leq .001$). From the data shown in Figure 7.8, it is clear that participants creating code with spreadsheet syntax were actually significantly faster than participants using numerical indexing. This is in addition to creating more correct solutions as previously presented.

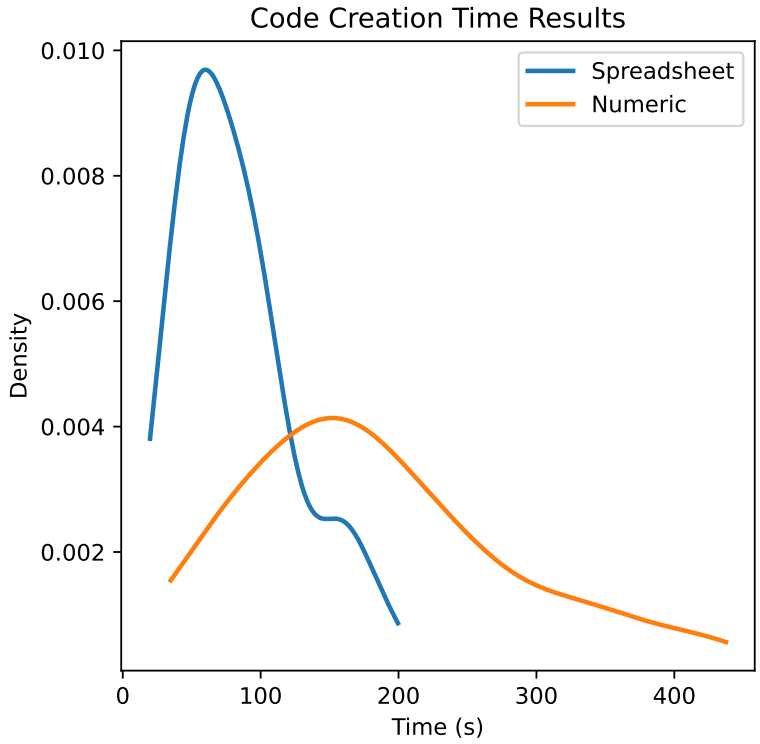


Figure 7.8: Kernel-density plot comparing time on task for spreadsheet syntax compared to numerical syntax (adapted from [18]).

One likely explanation for this effect is that participants considered the spreadsheet syntax more intuitive and spent less time reading documentation and verifying their solution. In contrast to the complex, programming-adjacent concepts needed to understand numerical indexing (zero indexing, identical syntax to access both dimensions instead of a clear separation into identifiers for rows and columns), the spreadsheet syntax could potentially be known to data practitioners from working with spreadsheet software during previous data engineering

projects.

7.4.4 Conclusion

In conclusion, aligning with the syntax that practitioners know from working with other tools should be one of the important considerations when designing a new DSL. More concretely, in the case of DSLs for data engineering by subject-matter experts, spreadsheet syntax is a promising alternative to more traditional programming syntax that should be further explored.

The improved correctness and speed up when writing code we described might be limited to smaller datasets (because the alphanumeric indexing of columns does not scale well). However, these improvements are highly relevant in open data contexts because most open datasets are small and provided in tabular formats [44, 57]. Especially, the improved correctness of program comprehension is an important improvement given the importance of understanding source code due to the increasing amount of code generated by artificial intelligence tools.

While we have shown a positive effect of using spreadsheet syntax and can discuss potential reasons based on our knowledge of the subject, more rigorous qualitative studies would be needed to describe the causes for these effects. In future work, we would like to extend the analysis of quantitative data by adding insights from experiment participants completing exit surveys.

8

Conclusion

In this thesis, we researched data engineering with open data and the challenges that must be resolved to enable open collaborative work to improve the overall data quality for every participant. We described the collaboration systems during data engineering and highlighted the importance of subject-matter experts who contribute their domain expertise to understand complex data correctly. We chose a problem we were well-positioned to solve, the lack of an adequate shared collaboration artifact for both software engineers and subject-matter experts, and investigated a potential solution by constructing a DSL with a test bed to evaluate it using design science research.

We showed that a DSL has the potential to be used for collaborative work with subject-matter experts. In addition, we identified the effects of major design decisions and discussed them to provide a basis for additional language iterations and future DSL designers. In this chapter, we summarize important results and discuss them in the context of the initial research questions. We also provide an outline for future research.

8.1 Contributions

Regarding artifacts, we contributed to the design and development of Jayvee, a DSL for the creation of data pipelines. The implementation of Jayvee is available for users as an open-source project and can be used by practitioners to improve their data engineering. Additionally, to support the design, we developed a process for the iterative generation and evaluation of hypotheses of important language design decisions using controlled experiments with student participants. With the creation of an adaptable language and tools to run experiments, the evaluation studies can be extended to data practitioners from industry and further hypotheses in the future.

The scientific contributions of our work lie in accurately describing collaboration systems in data engineering, identifying important problems, and evaluating DSL design decisions for collaboration with subject-matter experts. These contributions are best discussed in the context of the initial research questions from section 3.1.

Research question 1 - how open collaborative data engineering with open data can be enabled - focused on the exploration of how data engineers collaborate and what challenges they face, was addressed during the problem identification and objective definition activities described in chapter 4. After investigating the open data ecosystem, we found that different stakeholders exist who can be categorized into the major categories of data publishers, infomediaries, and data consumers.

Unique to the open data context is that data publishers are not necessarily data consumers themselves and instead can be participants without an active interest or involvement, because they are only releasing data due to legal requirements. Instead, open data consumers must self-organize and improve data from sources that are not receptive to changes requested by the community. Due to this, open collaborative workflows are a way for the community to reduce individual workloads, and creating automated data pipelines that consistently improve data releases solves the challenges of data that is repeatedly released with errors.

While describing the wider social systems in data engineering projects in more detail, we observed at its core a so-called project group which included the roles of mediator, software engineer, data engineer, and subject-matter expert. The involvement and challenges of non-technical contributors are an important element of these collaborations. The expertise of subject-matter experts is needed to correctly understand datasets, especially in complex domains such as open science. Our research showed that, while technical problems exist, social challenges play a major role in collaborative data engineering projects. Partially, these arise for the larger community, like the need for deep knowledge and specialized skills, but existing barriers to entry and difficulty in finding experts. But they are also relevant for smaller-scale collaborations in project groups, where the individual participants have no shared collaboration artifact or standard processes and lack specialized tools that support their work well.

As concrete recommendations, we therefore suggest first defining a standard collaboration artifact for the creation of data pipelines, adapting proven open collaboration workflows to data engineering, and providing a centralized project forge to disseminate these standards. Participants in open collaborative data engineering should be supported by purpose-built tools. In the context of the wider open data ecosystem, the creation of data communities should be supported by providing and aligning the incentives of all participants and highlighting models of working [17].

Because our background in software engineering allows us to contribute well-built tools, and we consider the missing standard collaboration artifact to be an important challenge, we focused our investigation on determining that a text-based DSL is a viable foundation for this collaboration artifact in open collaborative data engineering to answer RQ2. After the construction of a DSL using the pipes-and-filters architectural style, we demonstrated that it is possible to use it to solve existing, real-world data engineering problems. Additionally, our empirical validation with students as proxies for data practitioners has shown that a DSL is a valid alternative to GPLs to collaboratively develop data pipelines.

By using a DSL to solve data engineering exercises based on real open data sources, students

were able to experience improvements across speed, correctness, and the impression of difficulty [20]. Narrowing the investigations to specific features and uses of the DSL showed that the improved speed does depend on the activity under consideration. Code understanding was generally completed in similar times compared to a GPL with libraries. However, individual activities and feature combinations, such as code creation with spreadsheet syntax, took less time. It seems that potential speed improvements can be realized using DSLs for data pipeline creation, but how and to what extent requires more research.

In contrast, the correctness of solutions saw significant, consistent improvements when using a DSL or domain-specific syntax instead of a GPL with libraries. Our studies documented that students can understand data pipeline structures more correctly using a DSL based on pipes-and-filters [19]. Additionally, they could read and write code more correctly using spreadsheet syntax compared to numeric cell selection approaches [18]. These results show that a major advantage of using domain-specific concepts during data engineering with subject-matter experts can be found in improved correctness, but not necessarily faster implementation. In the context of our studies of data engineering for smaller-scale open datasets, a DSL is a valid basis for a collaboration artifact for subject-matter experts.

In research question 3, we asked which considerations are important for a DSL used to create data pipelines by subject-matter experts. Based on the qualitative user feedback we gathered as part of our evaluation, we described a number of effects experienced by participants when solving data engineering tasks with a DSL. Overall, the pipes-and-filters architecture we chose for the structure of the DSL has proven to be a good fit. It matches the common mental and visual model of data pipelines as DAGs, with blocks of computation connected by arrows. Our results show that how well the language aligns with the mental model for data pipelines of the users is very important for the ease of use when working with it. The core challenge is to find the correct level of abstraction from the technical details to what the users expect to see. When this level of abstraction is too low, users are confused by technical steps, but if it is too high, the language lacks the flexibility needed to work with low-quality data. Language

designers should be prepared to conduct empirical studies with their target audience in order to learn about their mental model of the problem domain.

One main feature that was universally described as improving understanding and being helpful is the separated pipeline overview that is created in any Jayvee pipeline model when blocks are connected using the arrow syntax. Even though it is possible to recreate this structure using a GPL, the strongly enforced and consistent structure found in Jayvee models helped users to orient themselves quickly and ensured that created models followed it, even without knowledge of good programming practices. Additionally, the structure was considered a helpful guideline and a reminder for the necessary implementation steps. From these descriptions, it seems clear that a strict structure without many options, ideally with a syntactically separated overview section, should be considered important for future DSL developments for users without a technical background.

Regarding collaboration and code reuse, the co-location of related code in a block is an upside of the pipes-and-filters structure. Additionally, providing everything needed to create a data pipeline from one language implementation and a standard library, instead of third-party libraries, has a positive effect on how easy the DSL is to understand, as users only need to comprehend one consistent glossary and one approach to solve problems. However, these upsides come with tradeoffs since excluding third-party libraries means more individual implementation work and a lower innovation speed.

An important consideration for DSLs used for data engineering is how to verify that data matches a schema. In our evaluation, participants considered user-defined value types easier to understand than callback functions used to filter datasets. Moreover, user-defined value types are described as enabling collaboration due to better code reuse. Despite these facts, the concrete design choice we made for Jayvee by implementing value types based on a collection of constraints was considered not intuitive. Future language designers (and future work) should consider value types as a core language concept and explore the best way to implement them.

On a more narrow level, we found that domain-specific syntax, such as the spreadsheet-inspired cell selection we evaluated for data engineering, can improve performance for subject-matter experts. Similar to DSLs, domain-specific syntax choices should also be investigated individually because their effect can change depending on the domain and target user. In our experience, understanding the previous experience of non-technical users is important knowledge that should be taken into consideration as soon as possible. A well-designed DSL should allow subject-matter experts to reuse their expertise from non-technical domains when implementing models. This way, their previous experience matters when learning and using the DSL, making them more productive faster.

When it comes to data engineering, for example, many subject-matter experts have previous experience working with spreadsheet software. This might be the case due to using it as a regular part of their job to accomplish unrelated goals or because they are actually cleaning and transforming data using spreadsheet software in one-off projects. DSLs for data engineering can make this a strength by reusing syntax and concepts from spreadsheet software.

Finally, empirical evaluations matter and are important for DSLs. Over the course of this work, we gained new insights that we did not anticipate and could not have formally deduced. Examples include feedback for the mental model and correct abstraction level, but also hearing about subject-matter experts learning data engineering from spreadsheet software. Executing evaluation studies with human participants was complicated to plan and execute, but provided unique insights. We recommend that any language designer include user studies in their design process.

8.2 Future Work

In future work, we want to improve the generalizability of the experimental evaluation by executing the same protocols with data practitioners from industry, as well as open data enthusiasts. Gaining initial data and insights from students as proxies for subject-matter experts in the open data ecosystem was valuable and allowed us to refine our experimental approach

and tool. Using this existing platform to run replication studies with participants from the real population would add additional rigor and deeper insights.

Further, it would be important to extend the experimental framework to other concrete features that emerged from the qualitative research designs during the evaluation, such as how to best implement value types. We learned that participants considered user-defined value types as easier to understand and a good basis for code sharing and collaboration; however, our design of using a collection of constraints on an existing value type was described as confusing. With additional mixed-method studies, we want to learn more about the effect of user-defined value types for collaborative data engineering and how to best implement them so that they are helpful for subject-matter experts as well.

Moving beyond the line of research presented here, we are interested in additional improvements of open collaborative data engineering workflows for open data. We identified and described multiple challenges from interviews with practitioners and shared multiple recommendations that we could not attempt as part of this work. Building or adapting a centralized collaboration platform to the specific needs of collaborative data engineering is a promising research and engineering challenge that would be important for setting and distributing collaboration standards among the open data community. Such a platform would connect potential contributors, match projects with experts, and provide standard tooling and processes for participants.

Finally, outside the possible engineering contributions, creating additional incentives and aligning existing ones for the provision of high-quality open datasets on a community level could improve data availability. This would require interdisciplinary work to identify existing incentives and work with government agencies and economists to create sustainable models that reward contributions to improving open data as a public good. As a result, higher-quality open data could be an even more important source of innovation and transparency for everyone.

References

[1] Ankica Barišić, Vasco Amaral, and Miguel Goulão. 2018. Usability driven DSL development with USE-ME. *Computer languages, systems & structures* 51 (Jan. 2018), 118–157. *https://doi.org/10.1016/j.cl.2017.06.005*

[2] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. 2014. DataHub: Collaborative Data Science & Dataset Version Management at Scale. arXiv:1409.0798 [cs.DB]

[3] Glenn A Bowen. 2008. Naturalistic inquiry and the saturation concept: a research note. *Qualitative research: QR* 8, 1 (Feb. 2008), 137–152. *https://doi.org/10.1177/1468794107085301*

[4] Virginia Braun and Victoria Clarke. 2012. Thematic analysis. In *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological*. American Psychological Association, Washington, 57–71. *https://doi.org/10.1037/13620-004*

[5] Raymond P L Buse, Caitlin Sadowski, and Westley Weimer. 2011. Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. ACM, New York, NY, USA, 643–656. *https://doi.org/10.1145/2048066.2048117*

[6] Maximilian Capraro and Dirk Riehle. 2016. Inner Source Definition, Benefits, and Challenges. *ACM Comput. Surv.* 49, 4 (Dec. 2016), 1–36. *https://doi.org/10.1145/2856821*

[7] Joohee Choi and Yla Tausczik. 2017. Characteristics of collaboration in the emerging practice of open data analysis. In *Proceedings of the 2017 ACM Conference on Computer*

Supported Cooperative Work and Social Computing. ACM, New York, NY, USA, 835–846. *https://doi.org/10.1145/2998181.2998265*

[8] Pablo Cingolani, Rob Sladek, and Mathieu Blanchette. 2015. BigDataScript: a scripting language for data pipelines. *Bioinformatics (Oxford, England)* 31, 1 (Jan. 2015), 10–16. *https://doi.org/10.1093/bioinformatics/btu595*

[9] Michael R Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole Goble, and The CWL Community. 2022. Methods included: standardizing computational reuse and portability with the Common Workflow Language. *Commun. ACM* 65, 6 (June 2022), 54–63. *https://doi.org/10.1145/3486897*

[10] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1277–1286. *https://doi.org/10.1145/2145204.2145396*

[11] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2018. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering* 23, 1 (Feb. 2018), 452–489. *https://doi.org/10.1007/s10664-017-9523-3*

[12] Nelson Fonseca, Joao Paulo Fernandes, Mario Pires, and Simao Melo de Sousa. 2020. PACE: A DSL-based approach to manage complex build pipelines. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 43–50. *https://doi.org/10.1109/seaa51224.2020.00018*

[13] Martin Fowler and Rebecca Parsons. 2010. *Domain-Specific Languages*. Addison-Wesley Educational, Boston, MA.

[14] David Garlan and Mary Shaw. 1993. AN INTRODUCTION TO SOFTWARE ARCHITECTURE. In *Advances in Software Engineering and Knowledge Engineering*.

Series on Software Engineering and Knowledge Engineering, Vol. 2. WORLD SCIENTIFIC, 1–39. *https://doi.org/10.1142/9789812798039_0001*

[15] Saul Greenberg and Bill Buxton. 2008. Usability evaluation considered harmful (some of the time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. Association for Computing Machinery, New York, NY, USA, 111–120. *https://doi.org/10.1145/1357054.1357074*

[16] Philip Heltweg and Dirk Riehle. 2023. Challenges to Open Collaborative Data Engineering. In *Proceedings of the 56th Hawaii International Conference on System Sciences* (Hyatt Regency Maui), Tung X Bui (Ed.). 679–688.

[17] Philip Heltweg and Dirk Riehle. 2023. A Systematic Analysis of Problems in Open Collaborative Data Engineering. *ACM transactions on social computing* 6, 3-4 (Oct. 2023), 1–30. *https://doi.org/10.1145/3629040*

[18] Philip Heltweg, Dirk Riehle, and Georg-Daniel Schwarz. 2025. Is spreadsheet syntax better than numeric indexing for cell selection? arXiv:2505.23296 [cs.PL] *https://arxiv.org/abs/2505.23296*

[19] Philip Heltweg, Georg-Daniel Schwarz, and Dirk Riehle. 2025. Can a domain-specific language improve program structure comprehension of data pipelines? A mixed-methods study. arXiv:2505.16764 [cs.PL] *https://arxiv.org/abs/2505.16764*

[20] Philip Heltweg, Georg-Daniel Schwarz, Dirk Riehle, and Felix Quast. 2025. An empirical study on the effects of Jayvee, a domain-specific language for data engineering, on understanding data pipeline architectures. *Software: practice & experience* 55, 6 (June 2025), 1086–1105. *https://doi.org/10.1002/spe.3409*

[21] Benjamin Hoffmann, Neil Urquhart, Kevin Chalmers, and Michael Guckert. 2022. An empirical evaluation of a novel domain-specific language - modelling vehicle routing problems with Athos. *Empirical Software Engineer* 27, 7 (Sept. 2022), 180. *https://doi.org/10.1007/s10664-022-10210-w*

[22] Florian Häser, Michael Felderer, and Ruth Breu. 2016. Is business domain language support beneficial for creating test case specifications: A controlled experiment. *Information and software technology* 79 (Nov. 2016), 52–62. *https://doi.org/10.1016/j.infsof.2016.07.001*

[23] Stefan Höppner, Yves Haas, Matthias Tichy, and Katharina Juhnke. 2022. Advantages and disadvantages of (dedicated) model transformation languages: A qualitative interview study. *Empirical Software Engineer* 27, 6 (Nov. 2022), 1–71. *https://doi.org/10.1007/s10664-022-10194-7*

[24] Harrie Jansen. 2010. The Logic of Qualitative Survey Research and its Position in the Field of Social Research Methods. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* 11, 2 (2010). *https://doi.org/10.17169/fqs-11.2.1450*

[25] A Jedlitschka and D Pfahl. 2005. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005*. IEEE, 10 pp. *https://doi.org/10.1109/ISESE.2005.1541818*

[26] Arne N Johanson and Wilhelm Hasselbring. 2017. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering* 22 (2017), 2206–2236.

[27] R Burke Johnson, Anthony J Onwuegbuzie, and Lisa A Turner. 2007. Toward a Definition of Mixed Methods Research. *Journal of mixed methods research* 1, 2 (April 2007), 112–133. *https://doi.org/10.1177/1558689806298224*

[28] Hanna Kallio, Anna-Maija Pietilä, Martin Johnson, and Mari Kangasniemi. 2016. Systematic methodological review: developing a framework for a qualitative semi-structured interview guide. *Journal of advanced nursing* 72, 12 (Dec. 2016), 2954–2965. *https://doi.org/10.1111/jan.13031*

[29] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2014. Design Guidelines for Domain Specific Languages. arXiv:1409.2378 [cs.SE]

[30] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.

[31] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal Opinion Surveys. In *Guide to Advanced Empirical Software Engineering*, Forrest Shull, Janice Singer, and Dag I K Sjøberg (Eds.). Springer London, London, 63–92. *https://doi.org/10.1007/978-1-84800-044-5_3*

[32] Kai Klanten, Stefan Hanenberg, Stefan Gries, and Volker Gruhn. 2024. Readability of domain-specific languages: A controlled experiment comparing (declarative) inference rules with (imperative) java source code in programming language design. In *Proceedings of the 19th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, Setúbal, PT, 492–503. *https://doi.org/10.5220/0012857800003753*

[33] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (Feb. 2015), 110–141. *https://doi.org/10.1007/s10664-013-9279-3*

[34] Dimitrios S Kolovos, Richard F Paige, Tim Kelly, and Fiona A C Polack. 2006. Requirements for domain-specific languages. In *Proc. of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD)*, Vol. 2006. Nantes, France.

[35] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-Specific Languages: A Systematic Mapping Study. *Information and Software Technology* 71 (March 2016), 77–91. *https://doi.org/10.1016/j.infsof.2015.11.001*

[36] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. 2018. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering* 23, 5 (Oct. 2018), 2734–2763. *https://doi.org/10.1007/s10664-017-9593-2*

[37] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering* 17 (2012), 276–304.

[38] T Kosar, Nuno Oliveira, M Mernik, M Pereira, M Črepinšek, Daniela Carneiro da Cruz, and P Henriques. 2010. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems* 7 (2010), 247–264. *https://doi.org/10.2298/CSIS1002247K*

[39] Thomas D LaToza and André van der Hoek. 2016. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software* 33, 1 (Jan. 2016), 74–80. *https://doi.org/10.1109/MS.2016.12*

[40] Ulf Leser, Marcus Hilbrich, Claudia Draxl, Peter Eisert, Lars Grunske, Patrick Hostert, Dagmar Kainmüller, Odej Kao, Birte Kehr, Timo Kehrer, Christoph Koch, Volker Markl, Henning Meyerhenke, Tilmann Rabl, Alexander Reinefeld, Knut Reinert, Kerstin Ritter, Björn Scheuermann, Florian Schintke, Nicole Schweikardt, and Matthias Weidlich. 2021. The Collaborative Research Center FONDA. *Datenbank-Spektrum : Zeitschrift fur Datenbanktechnologie : Organ der Fachgruppe Datenbanken der Gesellschaft fur Informatik e.V* 21, 3 (Nov. 2021), 255–260. *https://doi.org/10.1007/s13222-021-00397-5*

[41] Jonnathan Lopes, Maicon Bernardino, Fábio Basso, and Elder Rodrigues. 2021. Textual-based DSL for conceptual database modeling: A controlled experiment. In *Anais do XXXVI Simpósio Brasileiro de Banco de Dados (SBBD 2021)*. Sociedade Brasileira de Computação - SBC, 169–180. *https://doi.org/10.5753/sbbd.2021.17875*

[42] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[43] Claudia Misale. 2017. *PiCo: A Domain-Specific Language for Data Analytics Pipelines*. Ph. D. Dissertation. University of Torino, Italy. *https://doi.org/10.5281/zenodo.579753*

[44] Johann Mitlohner, Sebastian Neumaier, Jurgen Umbrich, and Axel Polleres. 2016. Characteristics of Open Data CSV Files. In *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE. *https://doi.org/10.1109/obd.2016.18*

[45] L Nascimento, D L Viana, P A M Neto, Dhiego A O Martins, V Garcia, and S Meira. 2012. A Systematic Mapping Study on domain-Specific Languages. In *Int Conf Softw Eng Adv*. Xpert Publishing Services, Lisbon, PT, 179–187.

[46] Kevin O'Leary, Rob Gleasure, Philip O'Reilly, and Joseph Feller. 2020. Reviewing the Contributing Factors and Benefits of Distributed Collaboration. *Communications of the Association for Information Systems* 47, 1 (2020), 24. *https://doi.org/10.17705/1CAIS.04722*

[47] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems* 24, 3 (Dec. 2007), 45–77. *https://doi.org/10.2753/MIS0742-1222240302*

[48] Arie Purwanto, Anneke Zuiderwijk, and Marijn Janssen. 2020. Citizen engagement with open government data. *International journal of electronic government research* 16, 3 (July 2020), 1–25. *https://doi.org/10.4018/ijegr.2020070101*

[49] Vijay Janapa Reddi, Greg Diamos, Pete Warden, Peter Mattson, and David Kanter. 2021. Data Engineering for Everyone. arXiv:2102.11447 [cs.LG] *https://arxiv.org/abs/2102.11447*

[50] Dirk Riehle, John Ellenberger, Tamir Menahem, Boris Mikhailovski, Yuri Natchetoi, Barak Naveh, and Thomas Odenwald. 2009. Open Collaboration within Corporations Using Software Forges. *IEEE Software* 26, 2 (March 2009), 52–58. *https://doi.org/10.1109/ms.2009.44*

[51] Andrea Mocci Roberto Minelli and Michele Lanza. 2015. I know what you did last summer. In *2015 IEEE 23rd International Conference on Program Comprehension*, Vol. 3. 1–28.

[52] Ashraf Shaharudin, Bastiaan van Loenen, and Marijn Janssen. 2023. Towards a Common Definition of Open Data Intermediaries. *Digit. Gov.: Res. Pract.* 4, 2 (June 2023), 1–21. *https://doi.org/10.1145/3585537*

[53] Micah J Smith, Jürgen Cito, Kelvin Lu, and Kalyan Veeramachaneni. 2021. Enabling Collaborative Data Science Development with the Ballet Framework. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2 (Oct. 2021), 1–39. *https://doi.org/10.1145/3479575*

[54] Micah J Smith, Roy Wedge, and Kalyan Veeramachaneni. 2017. FeatureHub: Towards Collaborative Data Science. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 590–600. *https://doi.org/10.1109/dsaa.2017.66*

[55] Ignacio G Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. 2015. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *CIDR*.

[56] V A Thurmond. 2001. The point of triangulation. *Journal of nursing scholarship: an official publication of Sigma Theta Tau International Honor Society of Nursing / Sigma Theta Tau* 33, 3 (2001), 253–258. *https://doi.org/10.1111/j.1547-5069.2001.00253.x*

[57] Jurgen Umbrich, Sebastian Neumaier, and Axel Polleres. 2015. Quality assessment and evolution of open data portals. In *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, 404–411. *https://doi.org/10.1109/ficloud.2015.82*

[58] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2016. Crossover Designs in Software Engineering Experiments: Benefits and Perils. *IEEE Transactions on Software Engineering* 42, 2 (Feb. 2016), 120–135. *https://doi.org/10.1109/TSE.2015.2467378*

[59] Dakuo Wang, Justin D Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. 2019. Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW (Nov. 2019), 1–24. *https://doi.org/10.1145/3359313*

[60] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (Dec. 1945), 80. *https://doi.org/10.2307/3001968*

[61] David Wile. 2004. Lessons learned from real DSL experiments. *Science of computer programming* 51, 3 (June 2004), 265–290. *https://doi.org/10.1016/j.scico.2003.12.006*

[62] Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE '14, Article 38)*. Association for Computing Machinery, New York, NY, USA, 1–10. *https://doi.org/10.1145/2601248.2601268*

[63] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science + Business Media, Heidelberg, DE. *https://doi.org/10.1007/978-3-642-29044-2*

[64] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2024. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM computing surveys* 56, 4 (April 2024), 1–42. *https://doi.org/10.1145/3626522*

[65] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2018. Measuring program comprehension: A large-scale field study with professionals. *IEEE transactions on software engineering* 44, 10 (Oct. 2018), 951–976. *https://doi.org/10.1109/tse.2017.2734091*

[66] Amy X Zhang, Michael Muller, and Dakuo Wang. 2020. How do Data Science Workers Collaborate? Roles, Workflows, and Tools. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1 (May 2020), 1–23. *https://doi.org/10.1145/3392826*

[67] Anneke Zuiderwijk, Natalie Helbig, J Ramón Gil-García, and Marijn Janssen. 2014. Special issue on innovation through open data - A review of the state-of-the-art and an emerging research agenda: Guest editors' introduction. *Journal of theoretical and applied electronic commerce research* 9, 2 (May 2014), I–XIII. *https://doi.org/10.4067/S0718-18762014000200001*

[68] Anneke Zuiderwijk, Marijn Janssen, and Chris Davis. 2014. Innovation with open data: Essential elements of open data ecosystems. *Information polity* 19, 1,2 (June 2014), 17–33. *https://doi.org/10.3233/IP-140329*

[69] Anneke Zuiderwijk, Marijn Janssen, and Yogesh K Dwivedi. 2015. Acceptance and use predictors of open data technologies: Drawing upon the unified theory of acceptance and use of technology. *Government information quarterly* 32, 4 (Oct. 2015), 429–440. *https://doi.org/10.1016/j.giq.2015.09.005*

A

Paper 1: Challenges to Open Collaborative Data Engineering

In this appendix, the paper *Challenges to Open Collaborative Data Engineering* is reproduced in full. The article was originally published as:

Heltweg, P., & Riehle, D. (2023). Challenges to Open Collaborative Data Engineering. In T. X. Bui (Ed.), Proceedings of the 56th Hawaii International Conference on System Sciences (pp. 679–688). [16]

[1] *https://creativecommons.org/licenses/by-nc-nd/4.0/*

Challenges to Open Collaborative Data Engineering

Philip Heltweg
Friedrich-Alexander-Universität
Erlangen-Nürnberg
philip@heltweg.org

Dirk Riehle
Friedrich-Alexander-Universität
Erlangen-Nürnberg
dirk@riehle.org

Abstract

Open data is data that can be used, modified, and passed on, for free, similar to open-source software. Unlike open-source, however, there is little collaboration in open data engineering. We perform a systematic literature review of collaboration systems in open data, specifically for data engineering by users, taking place after data has been made available as open data. The results show that open data users perform a wide range of activities to acquire, understand, process and maintain data for their projects without established best practices or standardized tools for open collaboration. We identify and discuss technical, community, and process challenges to collaboration in data engineering for open data.

1. Introduction

Open data can be created, used, modified, and shared by anyone and therefore has the potential to be a driver of innovation. Still, research has shown that users face challenges when using open data. Part of these challenges are technical issues that make accessing the data hard. Additionally, the quality of data sources is often poor (Purwanto et al., 2020).

Data engineering, the activity and process of preparing data for use for a specific purpose, is costly and routinely consumes large parts of the budget for data science projects (Terrizzano et al., 2015). The importance of data engineering in open data is even larger because of the varying quality of data provided by publishers.

However, because open data can be freely modified and distributed, a community of users can share the work of making data easier to use. Open-source development has shown that individual costs can be lowered if separate parties collaborate on shared artifacts. This form of egalitarian, meritocratic, and self-organizing collaboration, called open collaboration (Riehle et al., 2009), naturally extends to open data. Similar to the open-source workflow, being able to share intermediate artifacts between projects would allow distributed communities of episodic volunteer contributors to collectively increase data quality, motivated by their own reuse. Open data users could collaboratively work to improve data for themselves after it has been published. By doing so, they would not have to rely on data publishers that might be slow to improve their data or have no incentives to provide data in a well-structured format.

In contrast to open-source software development, large-scale open collaboration seems to be uncommon in data engineering by open data users, with most open data projects being completed by small teams (Choi & Tausczik, 2017). It is unclear why open data users do not collaborate as much as open-source developers.

Virtual collaboration plays a major part during data engineering and participants make extensive use of asynchronous collaboration tools like GitHub, Slack, or Email (Choi & Tausczik, 2017; Zhang et al., 2020). Especially open data can be shared and improved among geographically distributed, virtual communities. However, reusing existing software and workflows that have been developed for software engineering does not optimally support data engineering activities. Tools that support virtual and collaborative work during other phases of the data science workflow have shown promise, for example during feature engineering (Smith et al., 2017) or for the creation of labeled data (Reddi et al., 2021). A recent publication by Smith et al., 2021 shows that open-source software development practices can be used during feature engineering as part of a machine learning pipeline. Understanding how open data users collaborate virtually during data engineering will be essential to create workflows and tools that are better adapted to the challenges they face.

Yet, academic research into open data has mainly focused on data publishers. If data engineering by users is described, it is usually seen as just a phase of a larger data science workflow. Therefore, data engineering, as performed by users of open data, is

often mentioned in the literature but not described in depth. To support large-scale open collaboration in data engineering across multiple projects, it is necessary to know the participants, their workflows, and the challenges they encounter in their individual projects. We asked the following research question to create an overview of the involved elements:

Research Question: *Which elements of collaboration systems for data engineering by open data users exist, and what are potential challenges?*

We contribute an overview of existing practices, participants, the tools they use, and artifacts open data users create in the course of data engineering collaboration. To do so, we conducted an exploratory literature review to identify the state-of-the-art workflows and processes in projects built on open data. Going beyond the identification of the current reported practices, we elicited the potential challenges to collaboration in open data engineering. Our contributions can be used as a basis for future research into workflow methods and improvements of supporting tools for open collaboration during data engineering.

This paper is structured as follows: First, we review related work in section 2. The research approach for the survey is presented in section 3. Results of the survey are summarized in section 4, followed by a description of their implications beyond the immediate findings in section 5. After a discussion of the limitations in section 6, we summarize the results and point out future research opportunities in section 7.

2. Related Work

To the best of our knowledge, there exist no reviews of how open data users collaborate in data engineering. Mainly, insight is gained across the whole data science workflow from surveys or interviews with data science practitioners, often in commercial settings. If open data is mentioned, the focus of publications is mostly on open data publishers and the work they need to do to provide data of adequate quality.

The challenges of data engineering are an active research topic in corporate environments. Terrizzano et al., 2015 describe what they call "Data Wrangling" at IBM, highlighting various barriers like privacy or technical issues to data usage. Also at IBM, Zhang et al., 2020 investigate collaboration of data science workers in a large company environment. They find data scientists are highly collaborative, work in small teams and with a variety of tools. However, they point out it is unclear if their results are generalizable outside of the specific corporate environment at IBM.

Previous work by Wang et al., 2019, while mainly focused on work practices of data scientists and their impressions of automated AI, includes a review of academic literature on what roles exist in data science teams and tools that are used during data science activities. They find interactive computing software like Jupyter Notebooks to be a widely used tool in companies like IBM and Netflix. At the same time, they also point out the problem of overly complex tools and missing features to include domain experts in data science teams.

In the context of open data, Choi and Tausczik, 2017 used interviews and survey responses to gain insight into collaboration during open data analysis. Their results show most collaboration happens in small, interdisciplinary groups that mainly build tools to make the use of data easier or reports based on new information from data. They identify that open data analysis is a new phenomenon that has yet to develop standardized norms and practices, as well as the lack of a centralized collaboration platform, as reasons for the fact that large-scale open collaboration is uncommon. While some participants used GitHub, Choi and Tausczik discuss that the platform might lack features and call for further research into how a platform could best support open data analysis.

Zuiderwijk et al., 2014 take a wider ecosystem perspective of open government data, including data publishers. Their work includes a systematic literature review to identify key elements that allow for the publication and use of open data across all stakeholders, including publishers. These elements include releasing data on the internet, being able to search for appropriate data, processing it, and finally using the data and providing feedback to publishers. Additionally, they point out the need for elements to integrate different tools and data sources.

3. Methods

We conducted a systematic literature review (SLR) according to Kitchenham (Kitchenham, 2004). During an initial pilot study, we identified the need for a review because most research in open data focuses on the activities of data publishers. If data engineering is discussed, it is only in the context of a larger data science process and often specific to a domain. To identify collaboration practices that are applicable to all open data users, we had to create an overview from the state-of-the-art literature.

3.1. Search Strategy

We defined an initial search strategy and refined it with information from the pilot study. The pilot study itself consisted of an iterative and broader approach

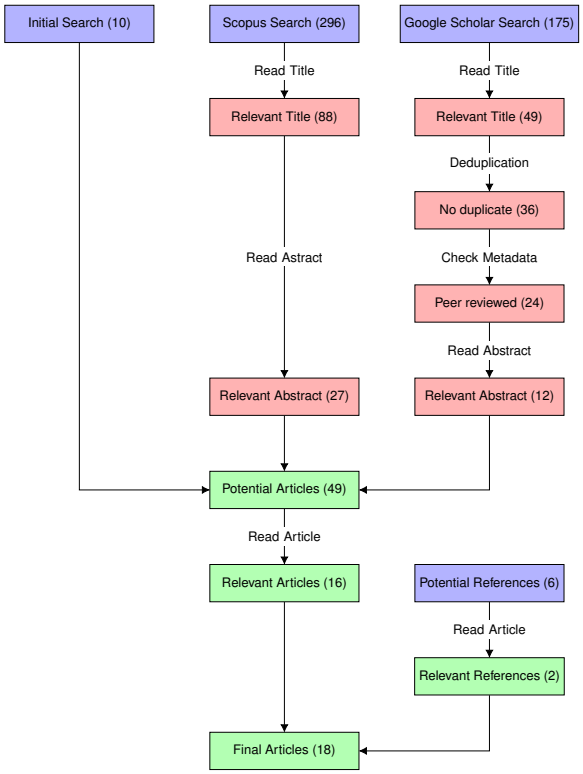


Figure 1. Process of the systematic literature review

to gain familiarity with the literature on collaborative work, data engineering, and open data over a variety of academic search engines. We included articles that were potentially relevant to the research question from these results. Based on the knowledge gained from the pilot study, we defined a systematic search strategy. We retrieved literature from Google Scholar and Scopus to cover a wide range of publications.

An overview of the process is shown in Figure 1. We searched for articles that included *open data* and *workflow*, *process*, *practices* or *participants* or variations thereof.

Scopus offers a comprehensive search interface that allowed us to search in the title and abstract of publications, while still ensuring relevance by making sure the keywords were not too far apart. The keywords in the search string used were:

```
("open data" OR "open-data")
W/5 ("workflow" OR "workflows"
OR "process" OR "processes"
OR "practices" OR "participants")
```

In addition, we limited the results to articles written in either English or German, the type of article only to journals or conference proceedings, and the publication stage to final.

The Google Scholar search was executed similarly. Because Google Scholar does not offer the ability to limit the distance of keywords in abstracts, we restricted the search to paper titles. The search string used was:

```
allintitle:workflow OR workflows OR
process OR processes OR practices
OR participants "open data"
```

For all searches, we only included articles published after 2008 because most publications on open data were created after that time (Purwanto et al., 2020).

We defined explicit inclusion and exclusion criteria:

- **Include** articles that describe data engineering workflows or processes with open data
- **Include** articles reporting on data engineering during a concrete project with open data
- **Exclude** articles that are not peer-reviewed journal or conference papers
- **Exclude** articles exclusively on data publishers
- **Exclude** articles that could not be retrieved in full

Every result of our search algorithm was checked for relevance first by its title, then by its abstract, and finally by skimming the article's full text and applying the inclusion and exclusion criteria. We removed duplicates and any articles that could not be accessed.

During the reading of potential articles, we noted down references that were potentially relevant because they were mentioned in the context of data engineering by open data users. We included these references in the pool of potential articles and verified their relevance by applying the same inclusion and exclusion criteria as for other articles.

3.2. Data Extraction & Synthesis

Our data extraction strategy was aimed at listing all elements of collaboration systems. We set up four shared documents in a sheet management software to track any mention of an activity, participant, tool used, or artifact created during data engineering with open data. We worked iteratively: When an element was mentioned in the current article, we added it to the corresponding list and saved a reference to the source.

When presenting work that builds on open data, most authors do not focus on the exact data engineering

activities performed but on the final results. To capture the whole scope of data engineering by open data users, elements were included liberally if they were mentioned in an article, even if they were not part of the main contribution. After every article, we merged entries that had already been identified in previous articles and noted the number of new elements found.

We followed the descriptive data synthesis approach described in Kitchenham, 2004 to provide a broad overview of data engineering in open data, prioritizing including edge cases over a compact summary. We therefore explicitly kept any elements that were only mentioned in few articles but provided new insights to cover the whole breadth of the process.

We grouped elements only when including individual elements did not offer additional insight, mostly for mentioned tools. Here, we merged entries that mentioned different concrete tools of a common type without a clear distinction, but kept any concrete tools that were mentioned specifically. For example, we grouped various mentions of PHP, Python, Java, etc. in the context of implementing software tools into one *general purpose programming languages* entry but kept *Open Refine* as a specific tool because it was mentioned multiple times explicitly.

Descriptions and examples were added for the extracted elements to clarify their meaning, the full descriptions are part of the raw data[1].

In addition, it became apparent during data extraction that a large number of different activities are performed by open data users during data engineering. We considered it important to preserve the detailed separation because the data engineering process is seldom described in detail, especially in the context of open data. However, with the large number of activities identified, we felt it would be valuable to create groups for a clearer overview. We created these groups one by one after all activities had been extracted by considering the list of activities, their descriptions, and examples. Once we felt that every activity was assigned to a matching group, we stopped the addition of new groups.

In a final step, the SLR results were shared with an open data expert working in the domain of open transport data for a member check (see Table 1). Their feedback pointed out some additional activities and distinctions between artifacts but was overall positive and confirmed that they felt the data was complete.

3.3. Concluding the search

Because the goal of this study was to identify the diversity of elements in collaboration systems for open data engineering, we used theoretical saturation as the stopping criterion for the search. Theoretical saturation is considered to be reached when no new insights are gained by analyzing additional data (Bowen, 2008). During our iterative approach to data extraction, we counted the number of new elements added with every article. We considered the data adequate when we did not add any new elements for multiple additional articles.

[1] Available on Zenodo at https://doi.org/10.5281/zenodo.6598447

3.4. Quality Assurance

During writing, we regularly held peer debriefing sessions (Spall, 1998) to ensure the credibility of the results. We discussed qualitatively with two other researchers that were not working on the same research but had experience with the methods we used. Two review sessions were conducted. First, we presented the search strategy of the systematic literature review and the resulting articles, as well as the extracted data. Based on the feedback, we added additional detail to the methods description and discussion of results. In a final peer debriefing session, we focused on the challenges that were identified from the data and how to best present them.

After obtaining the results, the results were discussed with an expert that has practical experience working on multiple open data projects as a form of member checking (Guba, 1981). For this, we created a handout document describing the research goals and methods and asked if we either had identified any elements that should not be included or missed any elements that were part of their practical experience. Based on the comments from the open data expert we then revised the results slightly and explicitly asked if the data seemed complete to which the open data expert confirmed that they had no further comments.

	Method	Participants	Topic
#1	Peer Debriefing	2 Researchers	Search Strategy & Results
#2	Member Check	1 Open data expert	Results
#3	Peer Debriefing	2 Researchers	Identified challenges

Table 1. Feedback methods used

Table 1 shows an overview of feedback sessions, participants, and main topics.

4. Results

We first discuss the search results of the systematic literature review. In addition, the identified elements of open collaboration systems during data engineering by open data users are presented by their categories of participants, activities, tools, and artifacts. As described

in section 3, the list of activities also includes themes that were created by grouping related activities.

4.1. Search results

The search returned 296 results from Scopus and 175 from Google Scholar, as shown in Figure 1. We initially excluded articles by their title, leaving 88 results from Scopus and 49 from Google Scholar.

For results from Google Scholar, we removed duplicates and articles that were not peer-reviewed journal papers or conference proceedings, leading to the exclusion of 25 articles. We then read the abstracts of all remaining articles and kept any that sounded relevant to the research question. After this step, 27 results from Scopus as well as 12 results from Google Scholar were included. Together with the initial search, 49 potentially relevant articles were identified.

The remaining results were read in full and the inclusion/exclusion criteria were applied. During this step, 33 additional articles were excluded, largely because they did not cover data engineering but only later phases of the data science workflow.

In the process of reviewing articles, six potentially relevant references were noted down for future revision. The same inclusion- and exclusion criteria as for other articles were applied, excluding two as not peer-reviewed and two as not relevant. The remaining articles (Lnenicka & Komarkova, 2019; Magalhaes et al., 2013) were included in the final literature pool, but did not contribute newly identified elements.

In the end, we identified a selection of 18 relevant articles.

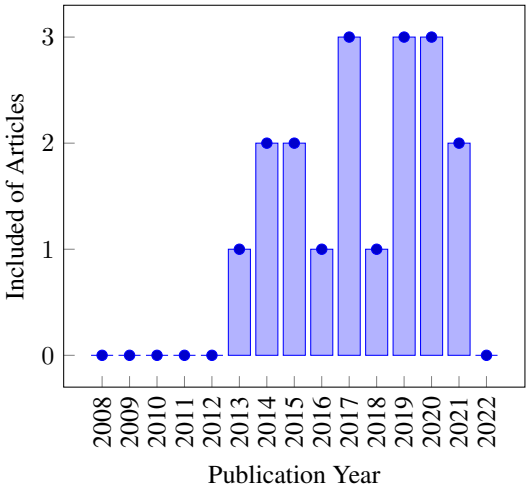


Figure 2. Publication date of included articles

We searched for publications starting in 2008, included articles were published between 2013 and 2021 (see Figure 2) with a slight increase since 2017. Most publications from 2022 could not be included because the original searches were performed during March and April 2022.

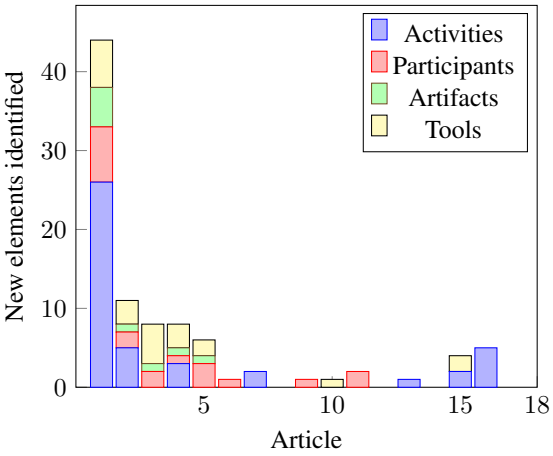


Figure 3. New elements identified by article

We tracked the number of new elements we added with every article (see Figure 3). Because the vast majority of elements were identified in the first articles and later articles contributed no new insights, we considered theoretical saturation to be reached and concluded the review.

The complete search results as well as data extraction are part of the raw data [1]. The raw data also includes sources for all identified elements of collaboration systems presented that have been omitted in the tables for readability.

4.2. Participants

Participants	
Businesses	Mediators
Citizen Scientists	NGOs
Civil Servants	Open Data Experts
Data Scientists	Organisations
Domain Experts	Private Citizens
Goverment Agencies	Researchers
Hackathon Participants	Software Developers
Infomediaries	Startups/Entrepreneurs
Journalists	Students
Legal Advisors	

Table 2. Participants in data engineering, by user role

A diverse group of users participates in collaboration systems for data engineering with open data, as shown in (Table 2). Open governmental data is a focus of academic research, so the list of participants is more detailed for the domain of public administration. It is noteworthy that government agencies and civil servants also act as users of open data, not only as publishers.

Open data allows interested parties insights into political processes, resulting in private citizens, NGOs, and journalists being common participants in open data engineering. These stakeholders are mainly involved in creating reports from data or tools for further insight.

We also identified commercial participants that use open data to build or enhance their products. These include large businesses like IBM that use open data as part of their larger data management strategy, but also startups that may build products based solely on open data. The literature also included references to intermediate entities, called Infomediaries, that offer services based on open data to end users, e.g., companies offering improved open data as a service, either by bundling it or providing processed data.

Open data users come from different backgrounds and view data from different perspectives. On one hand, the use of data creates a number of challenges in itself, requiring input from legal advisors and experts in open data or data science itself. On the other hand, working with open data is a technical challenge, which means software developers are often part of open data projects.

Depending on the context, understanding open data can be complicated and domain experts must be part of the data engineering process. This is especially true for the use of open data by researchers and students when the data might be part of a larger academic project, but also for citizen scientists that want to make sense of a complex problem.

In contrast to general data engineering, open data is often used by hobbyists or amateurs in the context of hackathons, private citizens, or students in university projects. Common to all these users is the low amount of organization and direction, an environment that should be ideal for open collaboration approaches.

4.3. Activities

A wide range of activities is potentially performed as part of data engineering by open data users. Table 3 shows an overview with the larger themes that emerged from the data. Not every activity will be executed during a given project, often only a small subset of activities is needed to make raw data available to use in an application.

In any case, users will need to perform activities related to *acquiring* and *assessing* open data to use. Most often, acquiring data takes the form of searching or discovering data and extracting it by downloading a data set. More complex projects might need to build infrastructure to automatically access data repeatedly. Not all data is easy to extract either, some data publishers require the creation of accounts or impose limits on how often data can be downloaded. After the data is acquired, it must be assessed for appropriate scope and legal compliance. To do so, users often visualize or preview part of the data. From expert feedback, we learned that availability is often a concern for open data users, when relying on an open data source it is important to verify that it will be consistently reachable. This process is necessarily iterative with backtracking whenever a data source lacks the content, license, or availability needed to be useful.

Once appropriate data has been acquired, it can be *improved* or *extended* with additional data. These steps contain a large number of technical activities like changing data format or structure, normalizing values, as well as finding and fixing errors. Additionally, users link data with other data sets and add metadata like data quality indicators. If the data is in a different language it might be required to translate it, either by employing automated translation tools or by hand. Activities that are preparing data for later stages in an ML workflow like feature creation and labeling of data could be considered project-specific and therefore not relevant to general data engineering. They are included here because well-structured, public data sets can be a useful basis for multiple ML projects that have no direct relation (Reddi et al., 2021).

For open data projects that are planned to exist long-term, *maintaining* data becomes a concern. Users need to write documentation about the process to make sure it can be repeated and data can be refreshed if it changes, like e.g. transportation schedules. Archival of open data might be necessary, especially if the underlying data source is unreliable or old data is replaced with new data.

To perform any of these activities it is essential to *understand* the data. This can be a purely technical challenge to learn the data format and structure of the data. Users analyze parts of the data or create small, ad-hoc experiments to gain insights into the data. Often, understanding data also requires understanding the underlying problem domain. Depending on the complexity of the context, this can mean having to ask (and find) domain experts or having to build up domain knowledge.

During all of these activities, open data users *communicate* with different participants in the

Acquire	Assess	Communicate	Extend	Improve	Maintain	Understand
Build Infrastructure	Ensure Anonymity	Ask Publisher	Add Metadata	Aggregate	Archive	Analyze
Discover	Evaluate	Discuss	Create Features	Clean	Document	Ask Experts
Extract	Preview	Find Community	Label	Combine	Refresh	Experiment
Read Documentation	Measure Availability	Find Skilled Users	Rate	Curate		Learn Domain Knowledge
Search	Verify License	Give Feedback	Translate	Enrich		Learn Structure
Select	Visualize / Plot Data	Request Data		Link		
Store		Share Data (Publisher)		Normalize		
Validate		Share Data (Stakeholders)		Reformat		
		Share Information		Repair		
				Structure		

Table 3. Activities performed during data engineering by open data users

ecosystem. They ask questions and provide feedback to data publishers, search for skilled users or domain experts in a community surrounding the data, and share their data and additional information with others. It is noteworthy that interactions with data publishers are expected and open data portals provide avenues to contact them. On the other hand, communicating with the larger community of users that are interested in the same data is less common during data engineering. Regarding other users, activities related to identifying experts and finding other community members are mentioned more often.

4.4. Tools and Artifacts

Tools used	
Auth Providers	Kaggle
Big Data Processing Tools	Notebooks
Blogs / Websites	Official Discussion Board
Command Line Tools	Open Data Repositories
Data Science Libraries	Open Refine
Databases	Sheet Software
Domain Specific Languages	Statistical Computing Languages
Domain Specific Software	Translation Software
General Purpose Languages	Travis
git	Visualization Tools
GitHub	Wikis

Table 4. Tools used during data engineering by open data users

Table 4 shows mentions of tools in literature. We could not identify a standard tool outside of Open Refine which was mentioned multiple times. Depending on the technical skills of project members, employed tools can be self-developed (e.g., based on general-purpose languages) or pre-made applications like Wikis or Sheet Software. After expert feedback, we added custom *Software Applications* as an explicit artifact. We previously assumed open data practitioners collaborate on building their own software in an open-source development process (so the artifact would be *Source Code*) but some applications (e.g., data validation tools) are also developed internally and only shared as closed-source programs.

Visualization tools play an important role in the data engineering workflow because they allow users to quickly evaluate data for quality and scope. When performing the more technical activities for acquiring and improving data, practitioners rely largely on general-purpose programming languages like Python or Java and the surrounding ecosystem of tools like Jupyter Notebooks and GitHub.

Open data repositories are mentioned often, but mainly just as a source of raw data. In contrast to open-source development, where collaboration increasingly happens on GitHub as a central project repository, many different open data repositories exist and data is spread between them. To find experience reports about data, documentation, and feedback, users must visit multiple, disconnected locations like publisher websites, practitioner blogs, or discussion boards.

Created Artifacts	
CI Definitions	Notebooks
Comments on Data	Processed Data
Data Quality Ratings	Raw Data
Documentation	Software Applications
Feedback-/Experience Reports	Source Code
Metadata	

Table 5. Created artifacts by open data users during data engineering

Similarly, open data practitioners do not collaborate on one well-defined, shared artifact. Various artifacts are created as part of data engineering activities (see Table 5) but they mostly are related to metadata or tools to deal with data.

5. Challenges

Open data has the potential for productive open collaboration because the data itself as well as any products resulting from it can be shared freely. Despite this, data engineering in open data is largely considered an activity for data publishers that concludes when the data is made public. We identified a number of potential challenges from the results of the systematic literature review:

- Need for specialized skills but high barriers to participation
- Finding and connecting with other community members
- No standard tools or artifacts
- No well-understood collaboration practices

First, the use of open data requires additional skills, making it more difficult for domain experts to participate. Experience with software development is a vital part of data engineering and software developers are common participants in open data projects. In addition, general-purpose programming languages, as well as statistical computing languages, were among the most mentioned tools for data engineering in our literature review. Aside from software engineering, the required data management skills can impose a barrier as well. Common formats to describe structured open data include semantic web formats like RDF, yet in a recent survey of researchers in Kjærgaard et al., 2020 only 7% of respondents said they were comfortable using it. The challenge will be to lower technical barriers to participation while at the same time staying flexible enough to work with a variety of data sources, formats, and qualities.

A second challenge is connecting members of open data communities. Our results show that the process of understanding data involves learning domain knowledge by finding domain experts or help from a community. Additionally, skilled users have to be identified to help with various barriers from software development to legal advice. Here, the challenge to open collaboration is that users must be able to identify and contact other participants that have a required skill set and are interested in the same data. Due to the public nature of open data, communities can naturally form around an area of interest but it is hard to find other members (Ruijer & Meijer, 2020). Other domains of open collaboration like open-source software development or wikis provide a central location (e.g., GitHub or Wikipedia) for community members to interact. In open data, a similar role could be accomplished by open data repositories, yet they are focused on providing data and less on building communities around common interests.

The lack of a standard artifact is an additional challenge for open collaboration. In open-source software development, community members collaborate on clearly defined artifacts, expressed in source code, like frameworks or libraries. These artifacts have in common that they are not competitively differentiating but instead get used to build additional, potentially closed-source products on top of. By collaborating on an underlying artifact, open-source developers can lower individual costs and with increased generality and quality of the artifact improve their individual applications. We could not identify a similar artifact in open data engineering. Most artifacts described in Table 5 are metadata surrounding the use of open data but not the data engineering steps themselves. It will be a challenge for open data engineering to find an intermediate artifact that is generic enough to be of use for many projects but can be developed collaboratively. Even though the processed open data is an obvious artifact that can be re-shared with the community, it is too static. As identified by Terrizzano et al., 2015, data must be regularly refreshed to be up-to-date, the same is true for regularly released data sets like open transport schedules. An ideal intermediate artifact would be able to cope with changing or newly released data.

Without a shared artifact to collaborate on, data engineering for open data faces the challenge of a fragmented tooling landscape. Currently, traditional software development tools like programming languages and GitHub are common in data engineering (see Table 4). As pointed out by Choi and Tausczik, 2017, these tools lack features that support collaboration on data specifically. Especially during evaluation, participants also use a variety of visualization tools and sheet software to preview the scope and quality of open data. For large-scale open collaboration, a centralized location to collaborate on an artifact will be important. In the open-source approach to software development, this role has been increasingly filled by GitHub for source code and module repositories like npm. Because a more data-focused, well-built project forge has yet to be created, GitHub is currently also used in many open data projects but is missing solutions for e.g., previewing and visualizing data.

As a result of the previous challenges, limited collaboration practices have been developed and adopted during data engineering on open data. Instead, a mindset of data publishers on one side and data users on the other side is common. Participants acquire data as-is and improve it for their own use-case but seldom

share the resulting data with the community. If data is released with errors or in inconvenient formats, users provide feedback and quality ratings to publishers but do not work together to improve the data for everyone. One exception are so-called *Infomediaries* (see Table 2) that offer additional services on top of existing data. This mindset difference is in contrast to the open-source approach to software development, where developers are collaborating on shared source code that then is used in their individual projects. While some data engineering activities like evaluating the scope of data might be project-specific, others like finding and fixing errors could be shared by interested parties. A final challenge to open data collaboration will be to identify which activities can be performed by aligned participants and develop collaboration workflows and practices for them.

6. Limitations

Our search was limited to Google Scholar/Scopus as well as by the language of articles, potentially missing out on relevant articles. We noted and reviewed relevant forward references from the original search results to increase our confidence in the completeness of the results. However, the articles identified in the literature search only include academic work while some papers also cited not peer-reviewed content. An additional search of practitioner literature would improve the depth of the review.

We performed descriptive data synthesis for the results of the systematic literature review. Without extracting quantitative data, we can not make statistical inferences about how common or important the identified elements of open collaboration systems in data engineering by open data users are. Given the research goal of identifying the diversity of elements, this was appropriate and allowed us to contribute a descriptive overview. At later points in time, with an extended search, other forms of qualitative data analysis could also be used. Ideally, quantitative data should be surveyed from open data practitioners instead.

A threat to validity in the form of bias could exist because large parts of the academic literature on open data is related to open government data. Because our goal is to identify as many elements as possible, and we are not attempting quantitative data synthesis, the threat is mitigated. However, the potential to miss elements from other domains exists. We used expert feedback from an open transport data practitioner Table 1 to increase our confidence that we captured the whole breadth of the data engineering process.

7. Conclusion

In summary, we set out to identify elements of collaboration systems for data engineering by open data users and point out potential challenges.

We have performed a systematic literature review and descriptive data synthesis to find elements of open collaboration systems in data engineering. Our results show that open data users come from many domains, with varying technical skills, and perform a large number of activities. We could find different tools and artifacts, but no standard practice of collaboration across open data engineering.

We identified a number of potential challenges to open collaboration in data engineering: High barriers to participation but the need for specialized skills, identifying and connecting with a larger community, the lack of standard tooling and artifacts as well as missing collaboration practices.

These challenges are especially relevant for large-scale, virtual collaboration that has the potential to be very effective in the context of open data projects. Working virtually with unknown members in a larger community exacerbates the identified challenges and makes standard tooling and practices even more important. Our results will be the basis for the development of an open collaboration workflow method and supporting tool that allows data engineers to collaborate in a geographically dispersed and asynchronous manner.

Additionally, we plan to extend our description of open collaboration systems in data engineering and verify the discussed challenges. To do so, we will conduct interviews with open data practitioners as well as industry partners in further work.

Acknowledgments

The authors would like to thank the open data expert for their feedback, both in informal talks as well as in reviewing the paper results. Additionally, the thoughtful comments from the anonymous reviewers helped us improve the paper.

References

Bowen, G. A. (2008). Naturalistic inquiry and the saturation concept: A research note. *Qualitative research*, *8*(1), 137–152.

Choi, J., & Tausczik, Y. (2017). Characteristics of collaboration in the emerging practice of open data analysis. *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*.

Guba, E. G. (1981). Criteria for assessing the trustworthiness of naturalistic inquiries. *ECTJ*, *29*(2), 75. https://doi.org/10.1007/BF02766777

Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, *33*(2004), 1–26.

Kjærgaard, M. B., Ardakanian, O., Carlucci, S., Dong, B., Firth, S. K., Gao, N., Huebner, G. M., Mahdavi, A., Rahaman, M. S., Salim, F. D., Sangogboye, F. C., Schwee, J. H., Wolosiuk, D., & Zhu, Y. (2020). Current practices and infrastructure for open data based research on occupant-centric design and operation of buildings. *Building and environment*, *177*(106848), 106848. https://doi.org/10.1016/j.buildenv.2020.106848

Lnenicka, M., & Komarkova, J. (2019). Big and open linked data analytics ecosystem: Theoretical background and essential elements. *Government information quarterly*, *36*(1), 129–144. https://doi.org/10.1016/j.giq.2018.11.004

Magalhaes, G., Roseira, C., & Strover, S. (2013). Open government data intermediaries: A terminology framework. *Proceedings of the 7th International Conference on Theory and Practice of Electronic Governance*, 330–333. https://doi.org/10.1145/2591888.2591947

Purwanto, A., Zuiderwijk, A., & Janssen, M. (2020). Citizen engagement with open government data. *International journal of electronic government research*, *16*(3), 1–25. https://doi.org/10.4018/ijegr.2020070101

Reddi, V. J., Diamos, G., Warden, P., Mattson, P., & Kanter, D. (2021). Data engineering for everyone. *CoRR*, *abs/2102.11447*. https://arxiv.org/abs/2102.11447

Riehle, D., Ellenberger, J., Menahem, T., Mikhailovski, B., Natchetoi, Y., Naveh, B., & Odenwald, T. (2009). Open collaboration within corporations using software forges. *IEEE Software*, *26*(2), 52–58. https://doi.org/10.1109/MS.2009.44

Ruijer, E., & Meijer, A. (2020). Open government data as an innovation process: Lessons from a living lab experiment. *Public performance & management review*, *43*(3), 613–635. https://doi.org/10.1080/15309576.2019.1568884

Smith, M. J., Cito, J., Lu, K., & Veeramachaneni, K. (2021). Enabling collaborative data science development with the ballet framework. *Proc. ACM Hum.-Comput. Interact.*, *5*(CSCW2), 1–39. https://doi.org/10.1145/3479575

Smith, M. J., Wedge, R., & Veeramachaneni, K. (2017). FeatureHub: Towards collaborative data science. *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, 590–600. https://doi.org/10.1109/DSAA.2017.66

Spall, S. (1998). Peer debriefing in qualitative research: Emerging operational models. *Qual. Inq.*, *4*(2), 280–292.

Terrizzano, I. G., Schwarz, P. M., Roth, M., & Colino, J. E. (2015). Data wrangling: The challenging yourney from the wild to the lake. *CIDR*.

Wang, D., Weisz, J. D., Muller, M., Ram, P., Geyer, W., Dugan, C., Tausczik, Y., Samulowitz, H., & Gray, A. (2019). Human-AI collaboration in data science: Exploring data scientists' perceptions of automated AI. *Proc. ACM Hum.-Comput. Interact.*, *3*(CSCW), 1–24. https://doi.org/10.1145/3359313

Zhang, A. X., Muller, M., & Wang, D. (2020). How do data science workers collaborate? roles, workflows, and tools. *Proc. ACM Hum.-Comput. Interact.*, *4*(CSCW1), 1–23. https://doi.org/10.1145/3392826

Zuiderwijk, A., Janssen, M., & Davis, C. (2014). Innovation with open data: Essential elements of open data ecosystems. *Information polity*.

B

Paper 2: A Systematic Analysis of Problems in Open Collaborative Data Engineering

In this appendix, the paper *A Systematic Analysis of Problems in Open Collaborative Data Engineering* is reproduced in full. The article was originally published as:

Heltweg, P., & Riehle, D. (2023). A Systematic Analysis of Problems in Open Collaborative Data Engineering. ACM Transactions on Social Computing, 6(3–4), 1–30. [17]

A Systematic Analysis of Problems in Open Collaborative Data Engineering

PHILIP HELTWEG and DIRK RIEHLE, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

Collaborative workflows are common in open-source software development. They reduce individual costs and improve the quality of work results. Open data shares many characteristics with open-source software as it can be used, modified, and redistributed by anyone, for free. However, in contrast to open-source software engineering, collaborative data engineering on open data lacks a shared understanding of processes, methods, and tools.

This article presents a systematic literature review of collaboration processes, methods, and tools in data engineering as performed by open data users. An additional interview study with practitioners confirms and enhances the findings and strengthens the resulting insights.

We find an ecosystem with heterogeneous participants and no standardized processes, methods, and tools. Participants face a variety of technical and social challenges during their work. Our work provides a structured overview of collaboration systems in open collaborative data engineering, enabling further research. Additionally, we contribute preliminary guidelines for successful open collaborative data engineering projects and recommendations to increase its adoption for open data ecosystems.

CCS Concepts: • **Human-centered computing** → **Collaborative and social computing systems and tools**; *Empirical studies in collaborative and social computing*; • **General and reference** → Surveys and overviews.

Additional Key Words and Phrases: collaboration, data engineering, open data

ACM Reference Format:
Philip Heltweg and Dirk Riehle. 2023. A Systematic Analysis of Problems in Open Collaborative Data Engineering. *ACM Trans. Soc. Comput.* 6, 3-4, Article 8 (December 2023), 30 pages. https://doi.org/10.1145/3629040

1 INTRODUCTION

Open data is data that can be used, modified, and shared, free of charge. However, using open data can be challenging for many reasons, including poor quality of data sources, uncommon and undefined formats and schemata, and lack of well-understood workflows and processes. Data engineering, the process of extracting, preparing and transforming the data into a usable format, is a labor-intensive and hence costly engineering activity [18, 26].

Open-source development has shown that collaborating on shared software artifacts can lower individual costs and improve quality. Similarly, sharing intermediate artifacts between data-projects could allow contributors to collectively increase the quality of the data they use, as demonstrated by Infomediaries that add value to data by preprocessing it for other consumers [31]. We make the natural assumption, that, due to its similarities with open-source software, open data can be improved in open collaborative workflows in which self-organizing, meritocratic and egalitarian communities of users contribute to a shared artifact [20]. The collaborative effort can be driven by the users' own motivation to improve the data for their reuse, eliminating the need to rely solely on data publishers, who may lack incentives to provide it in a well-structured format.

Authors' address: Philip Heltweg, philip@heltweg.org; Dirk Riehle, dirk@riehle.org, Friedrich-Alexander-Universität Erlangen-Nürnberg, Martensstr. 3, Erlangen, Bavaria, Germany, 91058.

However, while open collaboration is common in open-source software development, data engineering is typically project-specific and done by small teams. These teams often reuse tools and workflows from collaborative software engineering like GitHub that, while passable, lack features to specifically support collaborative data engineering [4].

Tools and processes made specifically for collaborative data engineering would be needed for best results. However, it is unclear which challenges they need to address. For other stages of the data science workflow, using more specific tools and workflows has shown promising effects [19, 22, 23].

It is therefore important to understand how data engineers collaborate. To do so, it is key to not only consider specific software, but also who participates in the collaboration, their workflows and how they interact. In this article, we investigate the larger context and refer to the combination of participants, their workflows, and tools as well as the artifacts they create as collaboration systems. Especially for open data, collaborative data engineering is a complex activity with participants from various backgrounds working together [7]. Social systems and their interactions with technical infrastructure are a large factor that creates challenges for data users. So far, academic research has focused on data publishers and the technical challenges of publishing good quality data. A comprehensive theory of collaborative data engineering by data users is missing.

To move towards such a theory, we answer the following research questions:

Research Question 1: *Which elements of collaboration systems for data engineering by open data users are described in literature?*

Research Question 2: *How and in which roles do participants in collaboration systems for data engineering interact socially?*

Research Question 3: *What are challenges to collaboration in data engineering and why?*

We contribute a descriptive overview of the elements of collaboration systems, during data engineering by data users. Elements include participants, activities they attempt during the data engineering process, the tools participants use and artifacts they create. Additionally, we describe the social systems that participants work in, highlighting the different roles they fulfil and how they interact with others during collaborative data engineering.

Furthermore, we contribute a list of challenges that data engineering practitioners face during collaboration with a rich description and insights from interviews. Based on these insights, we discuss guidelines that contribute to successful open collaborative data engineering projects. For practitioners and researchers in an open data context, we provide a list of recommendations to increase the adoption of open collaborative data engineering in their communities. Together, these contributions provide a starting point for a theory of open collaborative data engineering that can be extended in future research. The insights gathered about social systems and challenges experienced by practitioners can be used to define better requirements for future software tools that plan to support collaborative data engineering.

This paper extends previous work [7] with a qualitative survey among data engineering practitioners, with a focus on social challenges experienced during collaborative data engineering. In addition, guidelines and recommendations based on these insights are discussed.

The following structure will be used to present the work: Initially, related work is discussed in section 2. A description of the research design that was used to answer the research questions follows in section 3. The results are presented in section 4, followed by a discussion of their implications and suggestion of guidelines and recommendations in section 5. Potential limitations and how we attempted to mitigate them are shown in section 6, after which we provide a summary and outlook in section 7.

2 RELATED WORK

Collaborative work, especially distributed collaboration, has extensively been studied in the domain of software engineering. O'Leary et al. [17] provide an overview of distributed collaboration types, based on a systematic literature review. They point out that it is important to consider both technical as well as social contributing factors for successful distributed collaboration, and describe previously identified factors found in literature. We follow a similar approach, considering both technical and social elements of collaboration systems but with a much more narrow focus. The insights regarding open collaboration in data engineering presented in this article provide additional data on contributing factors in one specific domain.

In collaborative software engineering, researchers have explored approaches with increasingly larger numbers of participants and less imposed social structure, from distributed software development or global software development, over crowdsourcing approaches like hackathons to open-source software development.

Distributed software development or global software development is especially relevant in an enterprise context [10]. Challenges described include communication with remote colleagues and accessing expert knowledge [8]. In their structured review, Jiménez et al. [10] conclude that technological tools and processes must be adapted to the specific needs of an organization to reap the benefits of distributed software development. LaToza and van der Hoek [14] contribute a model to categorize crowdsourcing approaches like hackathons or open-source software development and come to a similar conclusion – the need to develop and adapt workflows for software development tasks. With its close relationship to crowdsourced software engineering approaches, open collaborative data engineering faces similar challenges and more insight into the underlying social dynamics and challenges is needed to develop more appropriate tools and workflows.

Open collaborative data engineering is most closely related to open-source software development. In both, organizational structure is not formally enforced but emerges from the community. Previous work analyzed raw data from public mailing lists [1] and software forges [29] to gather insights into community structures and roles. More recently, the structure of collaborative projects and resulting challenges to social interactions and software architecture have been studied empirically [3, 25]. In this article, we focus on a description of social interactions and challenges in open collaborative data engineering in the hope to enable similar work in the future. Nonetheless, we also contribute guidelines and recommendations based on insight from practitioners.

For open-source software development, project forges have proved essential to enable open collaboration, for example by providing standard tools and artifacts inside companies [20] or increasing awareness of community activity, as found on GitHub [5]. Due to its popularity, GitHub has a strong influence on collaboration workflows used in software engineering with its pull-based development flow becoming the de facto standard in open-source software development.

Data science has been a focus of academic activity recently, however, most publications that contribute insights about collaborative work focus on machine learning or data analysis, often in commercial settings. In open data contexts, publications that describe data engineering almost solely look at how data publishers can provide better quality. As far as we know, no reviews of how open data users collaborate during data engineering exists.

Workflows of data scientists and their impressions of automated AI are the focus of recent work by Wang et al. [28], but they also provide a review of academic literature about data science teams and tools they use. They conclude overly complex tools pose a barrier for subject-matter experts to participate in data science teams. Likewise for corporate settings, Terrizzano et al. [26] describe data engineering at IBM, including barriers to data usage. Zhang et al. [30] gather data on collaboration of data science workers in large companies by conducting an anonymous survey. Their results

show data scientists collaborate actively, work in small teams, and use a variety of tools. Because their work is based only on survey responses from employees at IBM, they are unsure if their results generalize outside of environments at large corporations. Our work builds on these studies by providing more data on collaboration by data users in different contexts, like open data and hackathons.

In the process of developing a collaborative framework for feature engineering, Smith et al. [22] identify the four main challenges as task management, tool mismatch, evaluation of contributions and maintaining infrastructure from literature and user studies. Whereas their work takes the perspective of supporting machine learning projects, the challenges identified in this article relate to collaboration during data engineering on open data without an ML focus.

Zuiderwijk et al. [31] review the literature about open data ecosystems, identifying important elements and their activities that contribute to successful open data publication and reuse. They describe scenarios of interactions across the ecosystem that include the release of data, search for data, processing, and use of data as well as providing feedback to publishers. While their work provides insights into the wider ecosystems that exist for open data, we focus only on the collaboration by data users themselves to take a more detailed viewpoint.

Collaboration during open data analysis is studied by Choi and Tausczik [4] using interviews and surveys. Participants work in small, interdisciplinary teams and create tools and reports based on data. The authors identify the need for further research about how platforms can best support open data analysis, as traditional hubs for software engineering like GitHub lack important features. By contributing challenges to collaborative data engineering, we support further research and the development of better tools towards that goal.

3 RESEARCH DESIGN

The need for a review was identified from an initial pilot study of the literature that revealed that collaboration during data engineering is seldom discussed. We approached the research questions with a two-step research design, first reviewing the existing literature and then using the acquired knowledge to interview practitioners about their experience.

Initially, we performed a systematic literature review (SLR) according to Kitchenham [12]. The pilot study revealed that the focus of academic literature is often on later stages of a data science project, like data analysis or machine learning. In the context of open data, research about collaboration practices between data users is rare, while data publishers are covered. We therefore concluded that a review of the existing literature on collaboration systems of data engineering by open data users would be needed to close this gap.

In a second step, we complemented the acquired insight from literature with a qualitative survey according to Jansen [9]. We aimed to create a description of social systems and the diversity of challenges to collaborative data engineering among practitioners. We gathered qualitative data by conducting semi-structured interviews, informed from the previous literature review. In addition to new insights, we asked interview participants if they had experienced challenges we had previously identified from literature to verify our results.

3.1 Systematic Literature Review

We performed a systematic literature review (SLR) according to Kitchenham [12] to answer RQ1 *Which elements of collaboration systems for data engineering by open data users are described in literature?* To do so, we defined a search strategy consisting of data sources, queries, inclusion, and exclusion criteria following a pilot study. From the pool of relevant literature, we extracted and synthesized elements of collaboration systems for data engineering by open data users.

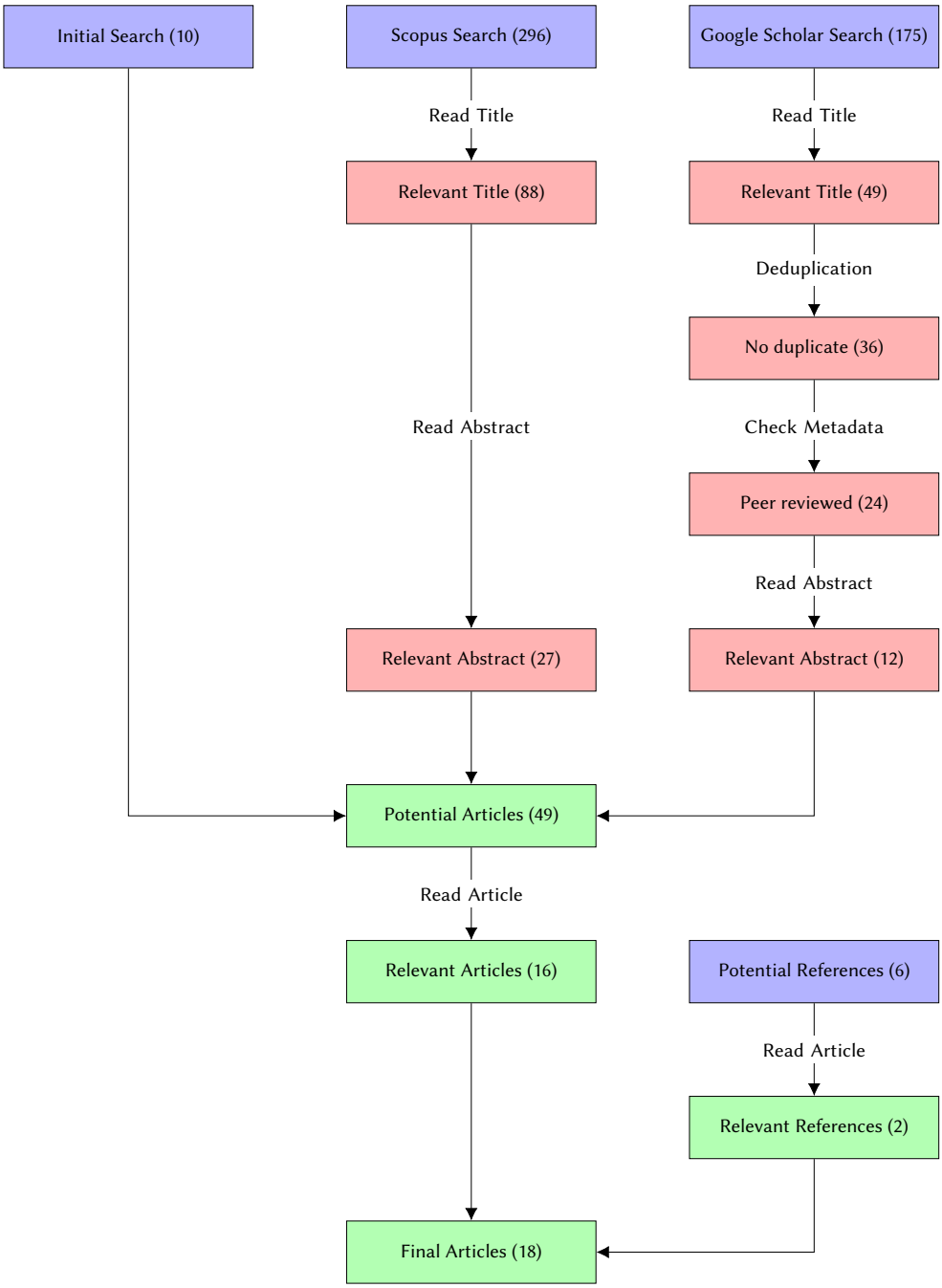


Fig. 1. Process of the systematic literature review

3.1.1 Search Strategy. An initial pilot study was performed to create the search strategy. For the pilot study, we iteratively looked at a broad range of literature from different academic databases to get an overview of existing research directions on collaborative work, data engineering and open data. We included the most relevant articles that were part of the pilot study directly in a pool of potential articles. From the results, we also decided on selecting Google Scholar and Scopus as sources for our literature search to include a wide range of publications because all relevant articles were found in these databases. We decided on searching for articles that include *open data* and *workflow*, *process*, *practices* or *participants* and variations of those terms. Most publications on

open data were published after 2008 [18], so we limited both searches to that time. An overview of the full search process is shown in Figure 1.

For Scopus, search results were restricted to journal articles or conference proceedings in English or German with a publication stage of final, leading to 296 results. We used the following search string in article titles and abstracts, making sure the keywords appear within five words of each other:

```
("open data" OR "open-data")
W/5 ("workflow" OR "workflows"
OR "process" OR "processes"
OR "practices" OR "participants")
```

Google Scholar does not offer the ability to enforce keywords to be close together. Searching the full text of articles returned many irrelevant articles, to narrow down the search and only include relevant results, we searched the title of publications with the following search string:

```
allintitle:workflow OR workflows OR
process OR processes OR practices
OR participants "open data"
```

The title-based Google Scholar search returned 175 articles.

After executing the initial search, we worked with 481 potential publications and ensured the results were relevant with several additional checks.

As a basis for decisions, we used the following inclusion and exclusion criteria:

- **Include** articles that describe data engineering workflows or processes with open data
- **Include** articles reporting on data engineering during a concrete project with open data
- **Exclude** articles that are not peer-reviewed journal or conference papers
- **Exclude** articles exclusively on data publishers
- **Exclude** articles that could not be retrieved in full

For both result sets, we excluded irrelevant papers based on their title and kept 88 results from Scopus and 49 from Google Scholar. Because Google Scholars results were less restricted, we also removed 13 not-peer reviewed articles and 12 duplicates from them.

Finally, we read the abstracts of every article and applied the inclusion and exclusion criteria to them to create a pool of 49 potentially relevant articles, 10 from the pilot search, 27 from Scopus and 12 from Google Scholar.

Next, we read all articles in full, noting down references that seemed especially relevant in the context of our research questions. After verifying that these were peer-reviewed and relevant, we included a further two articles [15, 16] from forward references.

Based on the inclusion and exclusion criteria applied to the full text of an article, we excluded a further 33 articles from the pool of potentially relevant articles, mainly because they focused on other phases of the data science lifecycle and did not include a description of data engineering.

This process led to a final pool of 18 relevant articles. We searched for articles published after 2008 but identified relevant articles between 2013 and 2021 (see Figure 2). The searches were executed during March and April 2022, meaning no article published in 2023 is included.

The search process, queries, and results are available in the published raw data [1].

[1] Available on Zenodo at https://doi.org/10.5281/zenodo.6598447

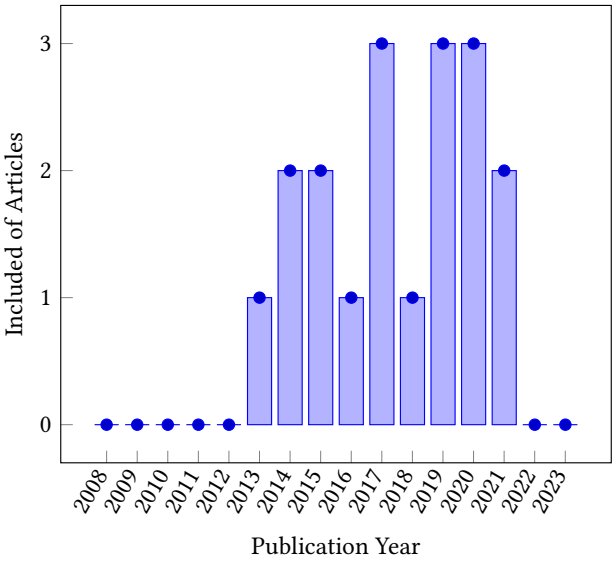


Fig. 2. Publication date of included articles

3.1.2 Data Extraction & Synthesis. We extracted data according to the descriptive data synthesis described by Kitchenham [12], using data extraction sheets for mentions of any activity, participant, tool used, or artifact created during data engineering with open data.

For every article, we noted any mention into the corresponding data sheet, merging any that were substantially similar to previously identified elements. Because publications that describe projects including open data often do not focus on data engineering, we included elements of collaboration systems liberally, even if they were not the main focus of the text. In the case of data engineering activities, we grouped activities into larger categories but report the individual activities separately as well. We created the categories of data engineering activities after data extraction, based on the list of activities, to include a detailed overview of the data engineering process.

We wrote descriptions and examples for all elements of collaboration systems that were identified during the data extraction and made the raw data available online[1].

Finally, we shared the results of the data extraction with a practitioner working on data engineering with open mobility data as a member check [6] (see Table 2). From their feedback, we added some activities and artifacts that were not described in literature. Overall, their feedback was positive, and they felt that the data was complete and aligned with their experiences.

3.1.3 Stopping Criteria. Theoretical saturation [2] was chosen as stopping criterion for the search because we wanted to identify the diversity of elements in collaboration systems for data engineering by open data users. Therefore, we tracked the number of new elements we added with every article (see Figure 3). We considered theoretical saturation reached when we did not gain any new insights after analyzing multiple articles and concluded the search.

3.2 Qualitative Survey

After gathering and analyzing the results from the systematic literature review, we extended the research design with an additional qualitative survey using semi-structured interviews according to Jansen [9]. Data was gathered from data engineering practitioners as a form of data source triangulation [27], allowing for insights outside academic literature. Jansen describes a typical

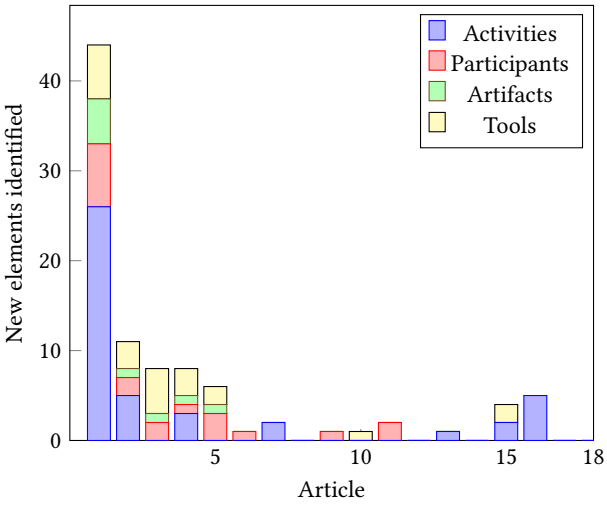


Fig. 3. New elements identified by article

empirical cycle of one-shot qualitative surveys as consisting of the definition of knowledge aims and sampling, data collection and finally analysis of the collected data.

3.2.1 Knowledge Aims & Sampling. Experiences of practitioners are needed to answer RQ2 *How and in which roles do participants in collaboration systems for data engineering interact socially?* and RQ3 *What are challenges to collaboration during data engineering and why?* We therefore aimed to create an inductive description of the diversity of social systems and challenges to collaboration in data engineering among people who have attempted collaborative data engineering before, informed by our previous knowledge from the structured literature review.

Pseudonym	Job Role	Employer	Project Types	Data Domains	Professional	Open Data
E1	Data acquisition	Academic publisher	Data curation	Material science	Yes	Both
E2	Software engineer	Software agency	Civic society	Transport	No	Yes
E6	Data engineer	Nonprofit organisation	Hackathons	Transport, Energy, Political	No	Yes
E8	Executive	Software agency	Hackathons	Geographical, Financial	Both	Yes
E9	Data engineer	Nonprofit organisation	Scientific	Medical, Biological	Yes	No

Table 1. Participants in semi-structured interviews

Because our goal was to describe the diversity and not to make statistical inferences, we decided to select a theoretically diverse sample. We reached out to a variety of contacts that worked on projects that included data engineering, and selected interview participants based on demographic data and attributes of the data engineering projects they typically attempt. An overview of participants and their attributes is shown in Table 1.

For a wide range of insights, we were able to sample perspectives from different job roles working directly with data, software, or management. Participants are employed in academia, industry or nonprofit organizations and work on diverse project types from data curation to hackathons. Typical data domains cover open data like transport or geographical, heavily regulated ones like medical or financial and complex scientific ones like material science.

Participants mostly work with open data in a non-professional context. However, we decided to also include projects that included collaborative data engineering on closed data or in professional contexts to gain additional insights.

3.2.2 Data Collection. We designed semi-structured interviews to collect qualitative data. We developed the interview guide according to the steps identified in Kallio et al. [11].

First, we identified the prerequisites for using semi-structured interviews based on the insights from literature during the SLR. This previous work helped us to conclude that data on roles and social interactions during collaborations was missing from the literature, but could be answered by asking practitioners. We also had already identified some challenges to collaborative data engineering that we could use to create the themes of the interview. Additionally, the data from the SLR also completed the second phase, retrieving and using previous knowledge.

Based on this knowledge, we developed an initial interview guide that was presented and discussed in internal testing with other researchers during a peer debriefing session (see Table 2). While interviewing participants, we always ended the interview with a question about topics we did not cover or should be asking about. With the feedback from participants, we continuously revised the interview guide. The final document is included in Appendix A.2.

The interview guide includes the main themes that we wanted to ask about to answer the research questions. These were:

(1) Demographic data
(2) The concept of collaborative data engineering itself
(3) Social systems in collaborative data engineering (Roles, interactions, and tools)
(4) Challenges to collaborative data engineering (Social, cultural, technical and previously identified challenges)

Additionally, every theme included some further question prompts to remind the interviewer of questions to ask. However, if the interview naturally flowed from a topic, we allowed for deviation from these detailed questions as long as all major themes were covered.

The interviews themselves were conducted electronically, over Zoom. We provided an interview handout describing research context, the interview process and how we would manage the resulting data to every interview participant a few weeks before the interview itself (the handout document can be read in Appendix A.1). We restated the interview process and important definitions of the research context before conducting each interview.

After an interview completed, we transcribed the recordings and replaced any personally identifying information with pseudonyms. The resulting transcript was then shared with the interviewee, asking them to correct any mistakes and provide a final acknowledgment of their consent to the transcript being used. As a result of interviewee feedback, we fixed some errors in tool names but made no changes to the content of interviews.

3.2.3 Data Analysis. As in the SLR, we used descriptive data synthesis according to Kitchenham [12] to analyze the interview transcripts. To do so, we set up data extraction categories:

(1) Social Systems
  (a) Roles
  (b) Interactions
(2) Challenges to collaborative data engineering
  (a) Social / Cultural Challenges
  (b) Technical Challenges
  (c) Previously Identified Challenges

With these categories in mind, we read through every transcript, highlighting and classifying sections from it according to the categories. We then combined similar segments into a topic with a brief description. This way, we arrived at a list of extracted topics and quotes from interview participants to support them.

In a final step, we further grouped the identified roles into three categories of project group, auxiliary roles and data community as we gained more understanding about the differences from interviews.

3.3 Quality Assurance

We employed peer debriefing sessions [24] to increase the credibility of the results. In these sessions, we discussed aspects of the research design and results with other researchers that had experience with the research methods used but were not involved in the topic or execution of the research itself.

	Method	Participants	Topic
#1	Peer Debriefing	2 Researchers	Search Strategy & Results
#2	Member Checking	1 Open data expert	SLR results
#3	Peer Debriefing	2 Researchers	Identified challenges
#4	Peer Debriefing	2 Researchers	Research design, interview study
#5	Peer Debriefing	2 Researchers	Interview process

Table 2. Feedback methods used

Participants and topics of the feedback sessions are shown in Table 2.

In the first peer debriefing, we presented the systematic literature review with a focus on the search strategy and initial results. From the feedback, we adapted our presentation to include more details about how we arrived at the final set of articles. In a second peer debriefing, we then discussed the challenges we had identified from the literature.

The follow-up interview study was also discussed in peer debriefings. We first presented our planned research design, including how it was informed by the previous literature review. In an additional meeting, we gathered feedback on the interview process itself from experienced interviewers to ensure we were conducting the interviews appropriately.

We presented the results of the literature review to an open data expert for their feedback about completeness as a member check [6]. For this, we created a handout document including the research context and asked if we had identified elements they thought were wrong or missed any important elements. Their feedback included some new elements, but there were no incorrectly identified elements. After adding the new elements, the expert confirmed they had no additional comments.

4 RESULTS

4.1 Elements of collaboration systems for data engineering described in literature

We extracted participants, activities, created artifacts, and tools used during data engineering by open data users from literature to answer RQ1, *which elements of collaboration systems for data engineering by open data users are described in literature?*

4.1.1 Participants. Working with data is a complex activity involving multiple skill sets. Therefore, participants in collaboration systems for data engineering come from various backgrounds, shown in Table 3, each contributing their expertise. Unsurprisingly, data scientists are often part of projects involving data engineering. When working with open data, open data experts can contribute knowledge about data sources or, together with legal advisors, help navigate the legal framework

Participants	
Businesses	Mediators
Citizen Scientists	NGOs
Civil Servants	Open Data Experts
Data Scientists	Organizations
Subject-matter Experts	Private Citizens
Government Agencies	Researchers
Hackathon Participants	Software Developers
Infomediaries	Startups/Entrepreneurs
Journalists	Students
Legal Advisors	

Table 3. Participants in data engineering

for data use. Lastly, making data usable is also an engineering challenge, which means software developers are an essential part of collaboration systems for data engineering.

Subject-matter experts play an important role in data projects, especially in the more complex problems that can be found in open science. Often, researchers are part of collaborative data engineering projects in the role of subject-matter experts. They help collaborators understand the meaning of data and assess data quality.

Commercial entities also participate, from large businesses that use open data to improve their existing products to startups that innovate with new applications using only open data. Depending on the company, participation in collaborative data engineering varies from active contribution to passive consumption of the final result.

A special position inside of open data ecosystems is taken by intermediate entities called infomediaries [31] that are located between open data producers and consumers and add value to data by processing it. These participants take in raw data and improve it for multiple downstream projects, a central part of open collaborative data engineering.

Besides commercial use, open data is primarily used in the context of open governments. Actors from public administration, like civil servants and government agencies, not only publish open data but also reuse data for their projects. Interested citizens interact with open data as journalists, as members of NGOs, or as students during Hackathons.

Common to the use cases of open data by students, citizen scientists or hackathon participants is a low amount of organization and direction, an environment that open collaboration could be productive in.

4.1.2 Activities. We could identify a large list of activities that are attempted as part of data engineering by open data users. All activities, as well as larger themes, are shown in Table 4. The overview includes all activities that were described as part of collaborative data engineering in the literature, but most projects only include a subset of activities.

At the start of any data-driven project, users must first source a usable data set. To do so, they perform an iterative cycle of activities related to *acquiring*, *understanding* and *assessing* data.

Activities related to the acquisition of data begin with data discovery, either organically or from directed search. Users have to extract data and store it in a system that is fit for their use case. Because data sources are not standardized or have download limits, extracting data often requires

Acquire	Assess	Communicate	Extend	Improve	Maintain	Understand
Build Infrastructure	Ensure Anonymity	Ask Publisher	Add Metadata	Aggregate	Archive	Analyze
Discover	Evaluate	Discuss	Create Features	Clean	Document	Ask Experts
Extract	Preview	Find Community	Label	Combine	Refresh	Experiment
Read Documentation	Measure Availability	Find Skilled Users	Rate	Curate		Learn subject-matter knowledge
Search	Verify License	Give Feedback	Translate	Enrich		Learn Structure
Select	Visualize / Plot Data	Request Data		Link		
Store		Share Data (Publisher)		Normalize		
Validate		Share Data (Stakeholders)		Reformat		
		Share Information		Repair		
				Structure		

Table 4. Activities performed during data engineering by open data users

reading provided documentation, building custom tools to interact with APIs and finding storage space.

Once data has been acquired, its usability has to be assessed. Dealing with licensing issues, making sure data is correctly anonymized, and verifying the data source has sufficient availability are common problems at this stage. Aside from technical and legal issues, data content and structure has to be understood to assess it usefulness for a project. Users engage in exploratory data analysis, using tools to preview data content (e.g., by plotting it) or data structure. Subject-matter knowledge is a requirement for working with more complex data sets and has to be either learned by the data users themselves or by finding and collaborating with subject-matter experts.

After acquiring, understanding and assessing data, data users process it, either by *improving* it or by *extending* it. During these activities, technical knowledge is required as users change data formats and structure, normalize values and fix errors. An activity that is challenging but adds a large amount of value to a data set is linking it with other sources.

Extending data most often takes the form of providing additional metadata, for example by writing usage reports or rating data sets on open data platforms. From expert feedback, we also included translating data as an activity, this can take the form of translating structural aspects like column names or content like the names of cities for open mobility data. If a data set is supposed to be the basis for machine learning projects, labeling data and creating features are common activities as well [19].

At the end of a collaborative data engineering process, activities related to the *maintenance* of the results, like archiving the resulting data, are required. An important but often neglected activity is the documentation of a project, learnings about data content and structure and the reasoning behind data engineering decisions. Some data domains like mobility deal with regularly updating data (e.g., public transport schedules that are released every few months) so data users must build infrastructure to refresh source data.

Underlying all these activities is *communication* with other participants. To date, concrete communication about a data set mostly flows from data users to data publishers in the form of questions, feedback or requests for more data. This interaction between data publishers and consumers is expected and supported by many open data portals. Direct communication among a larger data community is rarer and is mainly related to searching for other participants with a missing skill set or subject-matter knowledge.

4.1.3 Tools and Artifacts. We could not identify a standard tool used in collaborative data engineering among open data users. A summary of tools described in the literature is shown in Table 5. Noteworthy is Open Refine, which was mentioned multiple times.

Tools used by participants in collaborative data engineering range from self-developed using general-purpose programming languages to existing applications like sheet software or Wikis,

Tools used	
Auth Providers	Kaggle
Big Data Processing Tools	Notebooks
Blogs / Websites	Official Discussion Board
Command Line Tools	Open Data Repositories
Data Science Libraries	Open Refine
Databases	Sheet Software
Domain-Specific Languages	Statistical Computing Languages
Domain-Specific Software	Translation Software
General Purpose Languages	Travis
git	Visualization Tools
GitHub	Wikis

Table 5. Tools used during data engineering by open data users

depending on the technical skill level of project members. We included custom-made *Software Applications* from expert feedback as it was pointed out that collaborators not only develop open-source software but also share closed-source software applications like validation tools with the community.

As mentioned for the activities, understanding and assessing data often requires data exploration. Visualization tools provide a fast way to check data quality and content. More permanent solutions like automated data pipelines are developed using general-purpose programming languages or Jupyter notebooks with the help of classical software engineering infrastructure like git and GitHub.

Unlike project forges like GitHub for software engineering, open data portals play nearly no role in fostering collaboration among a community. They are mentioned often in the literature, but nearly always only as a data source and not as a platform to connect data projects. Whereas GitHub is the de facto standard to find software engineering projects that are open to collaboration, too many unrelated open data portals exist for any one of them to play a similar role.

Documentation of data projects, experience reports with data sets or expert advice is therefore scattered, and data users have to write blog posts, participate in discussion boards and read publisher websites.

Created Artifacts	
CI Definitions	Notebooks
Comments on Data	Processed Data
Data Quality Ratings	Raw Data
Documentation	Software Applications
Feedback-/Experience Reports	Source Code
Metadata	

Table 6. Created artifacts by open data users during data engineering

Similar to tools, no standard artifact exists that open data users collaborate on. Table 6 shows a summary of artifacts from the literature, most of which are metadata like comments or software artifacts to handle data. The processed and improved data itself was created as part of the data engineering process, but seldom shared with the community. Open data portals often do not enable

users to contribute any improvements to a data set back to the publisher, and tools like GitHub that are made to share source code are not well suited for sharing most data set formats.

4.2 Social systems in collaborative data engineering

We have identified roles and interactions from interviews with practitioners that work on data-driven projects that include collaborative data engineering to answer RQ2 *How and in which roles do participants in collaboration systems for data engineering interact socially?* Here, we initially present an overview of the roles and highlight essential interactions or those unique to collaborative data engineering in detail. As we were interested in a description of the diversity of roles and interactions, not every collaborative data engineering effort necessarily includes all roles and interactions mentioned here.

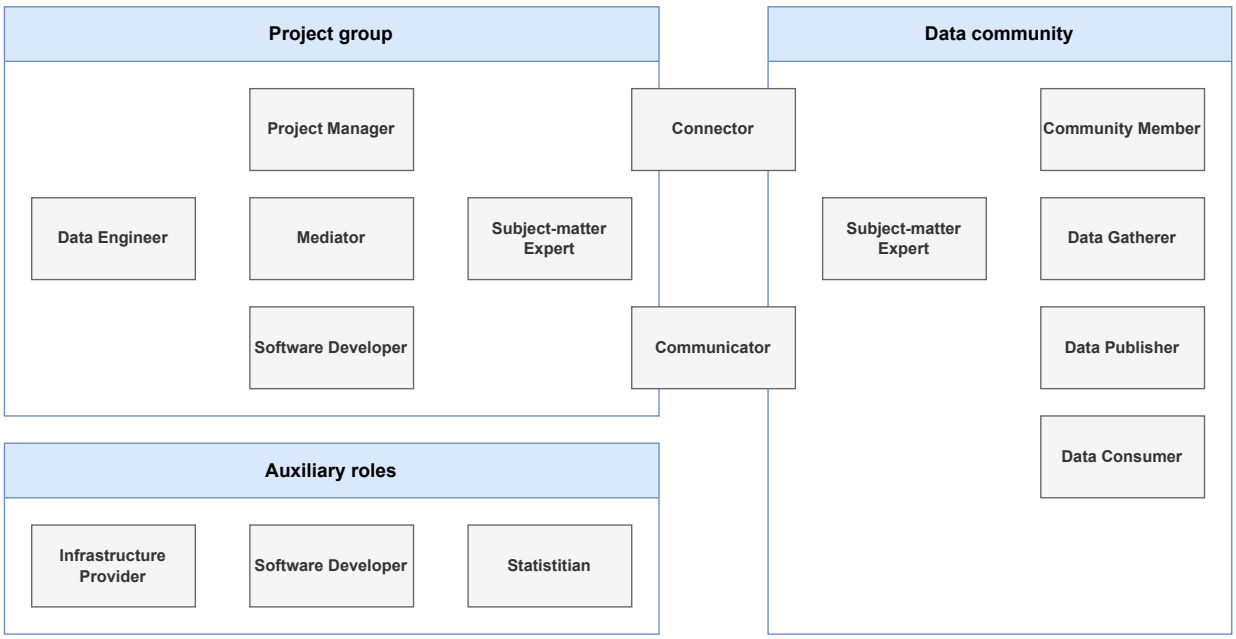


Fig. 4. Roles in collaborative data engineering as identified in interviews

When summarizing the roles that were described by interview participants, it became clear that collaborative data engineering interactions happen between three larger groups, as shown in Figure 4.

At the core, each data-driven project was attempted by members of a project group. This group is made up of participants that organize around a particular goal and data source, processing and publishing data towards archiving that goal.

Because of the interdisciplinary nature of data science, data processing inside the project group is driven by the three roles of data engineer, software developer and subject-matter expert. These roles provide their respective insights for working with the data, with the data engineer contributing knowledge about data formats and algorithms, the software developer providing infrastructure or technical requirements and the subject-matter expert explaining data meaning. Often, especially in smaller teams or in open data contexts, the role of data engineer and software developer are combined in a single contributor with a technical background.

Between these roles, a mediator must translate from the technical viewpoint to a subject-matter viewpoint and vice-versa. Mediation is a core interaction in collaborative data engineering projects, shown in Figure 5. A contributor assumes the role of a mediator and provides subject-matter experts

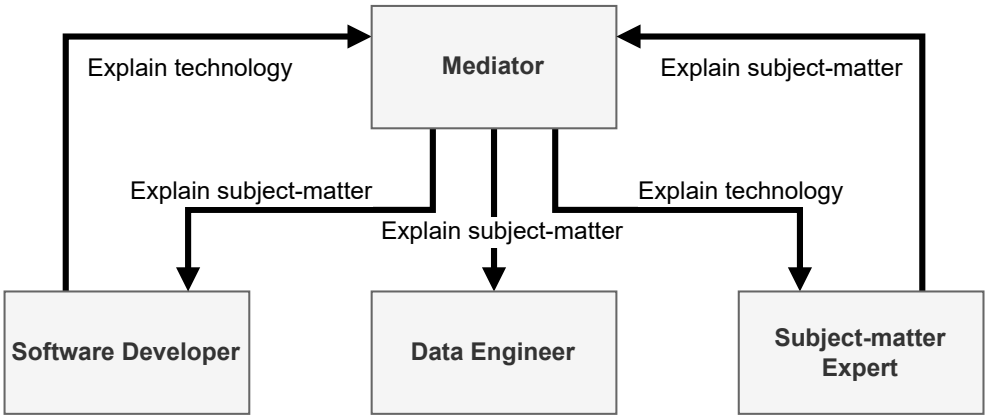


Fig. 5. Mediate during collaborative data engineering

with help in case of technical problems. On the other hand, they must explain the subject-matter to the other contributors, like software developers and data engineers. Often, the mediator role is filled by technical members of the team that, over time, learn enough from subject-matter experts to teach other technical contributors. In some projects, mediators are the subject-matter experts that get technical feedback from software engineers. As one example, E8 assumed the role of a mediator because they had a software development background and worked with subject-matter experts that had no technical experience: "It was the analog equivalents of them. So geographers, financial experts, but they don't have a lot of digital experience or can only scratch the thing at the surface. So there was a big gap between basically working from a really technical side to a really non-technical side." On the other hand, E1 saw part of their role in educating the data engineers on their team about the subject-matter due to their scientific background: "I just put people in touch with each other's teams and see if I can translate some of the science to the data people."

Finally, project teams include a project manager role. In commercial projects, this role does traditional project management work like defining a list of priorities and planning tasks. In comparison, open collaborative workflows rely on the self organization of participants. Therefore, the main contribution of a project manager role to open projects like hackathons is suggesting an idea and making sure participants align with it. E6, an experienced host of open data hackathons, calls the role 'idea owner' and describes it as: "[...] usually the person who pitches the challenge or proposes that use of that data set at the beginning of the event, but then sticks around and makes sure that the idea gets worked on."

Due to their collaborative nature, all data projects we have interviewed participants from are embedded in a larger data community that is not directly involved in the project but interested in the same data set or subject. Subject-matter experts can be part of the data community as well, and only periodically contribute to a collaborative data engineering effort without being part of the project group, either by finding the project by themselves or by someone from the project group reaching out to them.

Naturally, data publishers and data consumers of a specific dataset are members of this community. In some projects, data gatherers are also involved if a dataset is created from individually collected data points. Because all of these roles directly work with the data, they have a shared interest in collaboratively developing a consistent data schema, as shown in Figure 6. During this activity, project managers have to, on the one hand, stay in constant contact with data publishers and data consumers in the community (using connectors) and subject-matter experts and software developers on the other hand. Data publishers and software developers describe technical, availability or legal

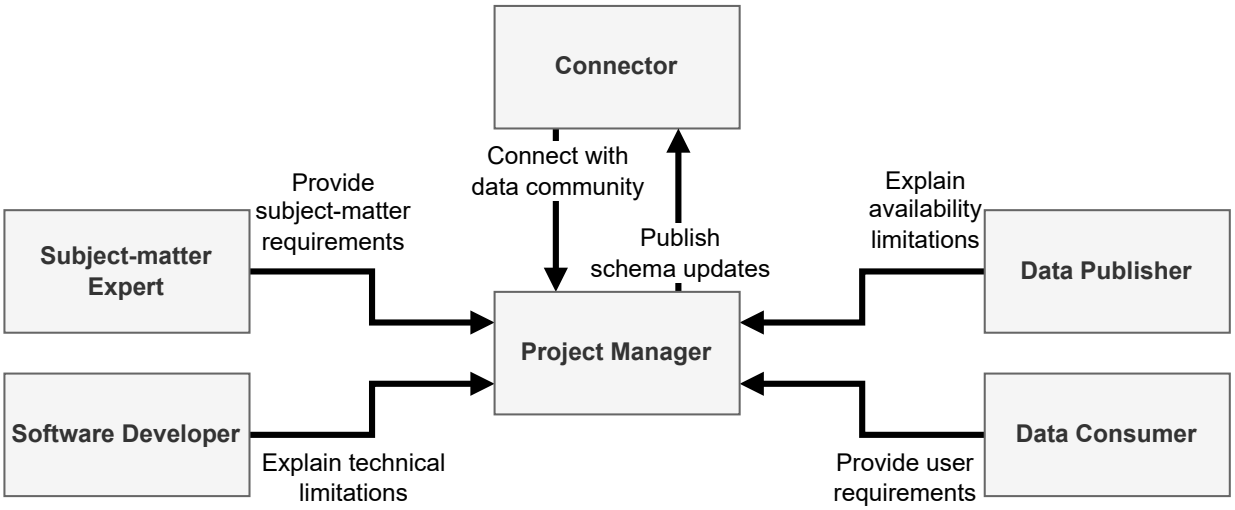


Fig. 6. Develop data schemas during collaborative data engineering

limitations to what and how data can be used. Inside those parameters, subject-matter experts can help formulate requirements for data schemas to capture the data domain adequately, while data consumers describe their projects and what information they are missing in the available data.

Additionally, in a broader sense, a data community is made up of individual community members that share a common understanding of data semantics and expectations towards data projects in their space. E6 expresses the concept of a data culture as follows: "I think there is something like the notion of a data culture. [...] Individual people grow up with more or less expectation around things like data privacy or just maybe rigor." Different data cultures can form around the same data set. Examples in the domain of open government data include the very rigorous, careful approaches of working with data by federal statistical offices, and the more playful, result-driven projects attempted by political activists during hackathons.

Bridging the gap between project groups and the larger data community are connectors and communicators. Whereas communicators describe and share the results of the work of project groups with the larger community, connectors actively bring together different roles in the social system.

The role of communicator can be performed by members of the project group itself, especially during hackathons, as E6 describes: "People who love visualizations and infographics are typically people who just communicate well, they would describe the the problem, the solution and the steps to get me to reach it using various media. Social media posts or illustrations [...]". Also performing this role are journalists who report about data projects. Because a report based on data is a potential downstream project for a data set, journalists also help to provide requirements and prioritization of what to work on. Occasionally, they can even contribute subject-matter expertise as E8 experienced: "Sometimes the journalists can help interpret the data, because they can help clarify the keys or what to look at for in the data."

In contrast to only sharing results with the data community, connectors work to bring together different entities in the ecosystem. In open collaborative projects, participants need to find each other to work together. Here, connectors are part of the ecosystem as networking organizations such as the Open Knowledge Foundation or data hubs like the German GovData.

In professional projects, connectors include data marketers and customer support employees that gather data use cases from customers. Prioritizing what data to work on and how a final data

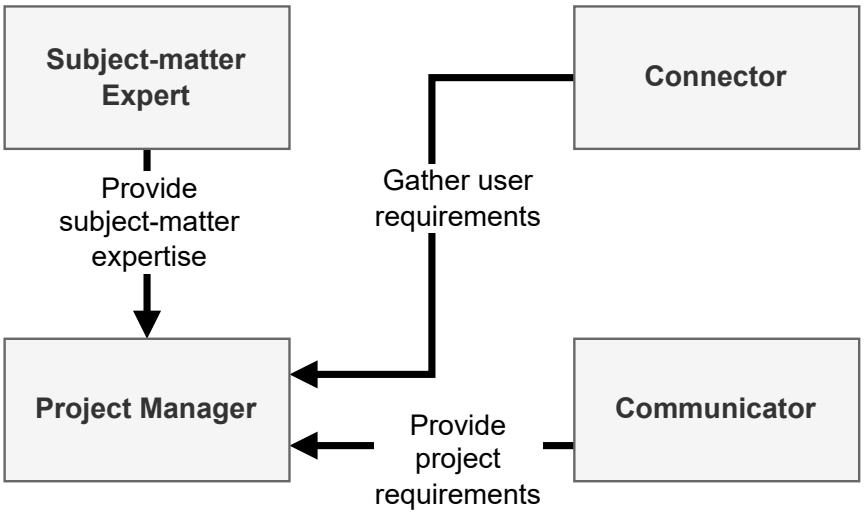


Fig. 7. Prioritize during collaborative data engineering

product should look like is an important interaction for these collaborative data engineering efforts. Figure 7 shows the different roles that interact during it. E1, who is working on a commercial data product, describes the need for insight from customers: "[...] everybody has a different need for that data set or the use case for a particular data set is so diverse. One person just wants it as a reference. [...] And then there's the whole AI/ML guys who want to use it to train models. [...] that's another thing we have to explore.". In their data project, these user requirements are gathered by data marketers. For hackathons with open data, requirements often arise from how the data is planned to be used by participants like journalists that fill a communicator role. In any case, subject-matter experts have to share their insight into what data is of high enough quality to be used and how important it is in the context of the planned use.

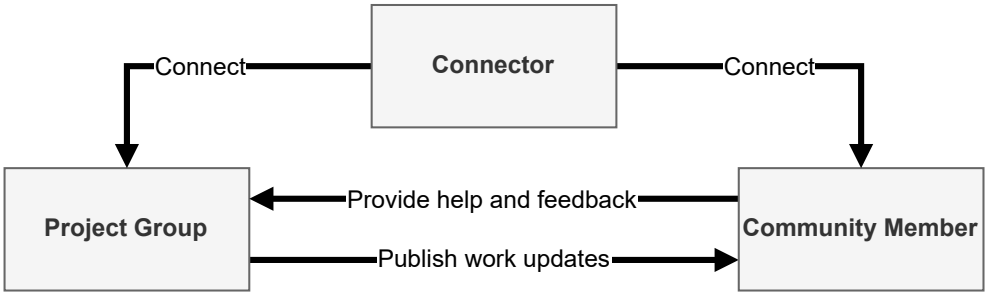


Fig. 8. Public work during collaborative data engineering

Increasingly common in open data projects is the concept of public work (see Figure 8). During public work, participants share updates about their progress and problems on social media or even livestream their work on services like TwitchTV. E6 mentions public work from their experience with open data hackathons: "People engaging in activities which they immediately communicate through some social media channel. Things like the way some people use LinkedIn or Twitter to announce very regularly their activities, their projects, their progress reports, all the way to things like streaming." This form of working is made possible by connectors that bring together interested members of the data community and members of a project group that are open to feedback or want

to increase the visibility of a data product. In the case of livestreaming, this form of working leads to a new way of gathering fast feedback and help from other contributors. E6 goes on to say: "So you have a channel where people post commentary. They drop in your channel to say, why are you doing this and not that?"

Finally, the project group is sometimes supported by auxiliary roles that provide services related to data engineering but do not participate directly in a data project. In open data contexts, these auxiliary roles are mostly related to providing infrastructure (like open data portals) or software developers that contribute open-source tools. In commercial settings, we have also encountered other, more highly specialized roles. As an example, due to the required rigor in the medical and biological data space that E9 works in, they are supported by statisticians: "[...] we have a statistician team. So, we are the data team and we send the data to the statistician team [...] they do the statistics.".

4.3 Challenges to collaborative data engineering

In this chapter, we highlight challenges that have either been described in the literature or been mentioned by interview participants to answer RQ 3, *What are challenges to collaboration in data engineering and why?* We report on challenges in three categories, previously identified challenges from literature, technical challenges from interviews and social challenges from interviews. As with roles and interactions (see subsection 4.2), the list of challenges highlighted here is not exhaustive but focuses on challenges that are either essential or unique to collaborative data engineering.

ID	Type	Title	Mentioned by
C1	SLR	Need for specialized skills but high barriers to participation	E1, E2, E8, E9
C2	SLR	Finding and connecting with community members	E1, E2, E6, E8
C3	SLR	No well-understood collaboration practices	E1, E2, E6, E8, E9
C4	SLR	No standard tools or artifacts	E1, E2, E6, E8
C5	Technical	Data representation	E1, E2, E6
C6	Technical	Inadequate tools	E2, E8, E9
C7	Technical	Infrastructure for data projects	E8
C8	Technical	Bad data sources	E1, E2, E8, E9
C9	Social	Conflicts with data publishers	E1, E2, E8
C10	Social	Unclear data use cases	E1, E2
C11	Social	Data semantics	E1, E2, E6
C12	Social	Missing incentives	E1, E6, E8
C13	Social	Missing knowledge	E1, E2, E6, E8, E9

Table 7. Challenges identified from literature and interviewee experiences

From the structured literature review, we identified four challenges [7], summarized in Table 7 with the type SLR. In all but one of the semi-structured interviews with practitioners, we explicitly asked for their experiences with the challenges and if they had also experienced them. For one interview, E9, we did not specifically ask for the challenges due to time constrains and a larger language barrier. Instead, we consider their confirmation of a challenge when they described a similar challenge they experienced themselves. Overall, the practitioners overwhelmingly experienced the previously identified challenges themselves.

C1, the need for specialized skills but high barriers to entry, could be confirmed by nearly every interview. Data engineering requires both subject-matter knowledge to understand data and

technical and data skills to work with it. This effect also explains the roles of data engineer, software developer and subject-matter expert that we described in subsection 4.2. It is a challenge to find participants to fulfil all those roles, especially in small teams or open data projects. In a survey among researchers, Kjærgaard et al. [13] found that only 7% of respondents were comfortable using RDF-files, meanwhile RDF is a standard data format for semantic web applications. One notable exception to the confirmation from interviews was E6, who disagreed with the need for specialized skills described in C1 because they host inclusive hackathons in which participants can try out unfamiliar roles and still contribute.

The challenge of finding and connecting with community members, C2, was also described in literature [21]. In this regard, open data projects lag other open collaboration ecosystems like open-source development or wiki content authoring. Contributing to this challenge is the fact that data portals are less frequented for data, as E8 describes: "For example, you found a software project, on GitHub, then it is really easy to collaborate. You can just open issues, fork it, contact the original maintainers of the project. But when you're on data portals for example, it is much harder because they are less frequented. You don't know who posted the data, maybe it was some kind of government agency or anything like that. So there is no one who feels responsible." Additionally, data portals are focussed on hosting data sets and are missing most social features of other project forges like GitHub or Wikipedia, making the discovery of other users hard. E8 goes on to elaborate: "And the users of the data sets are not visible anywhere. So that's a problem on the data portals, at least I know. It is actually really hard to get into contact with people that are working on it."

Well-understood collaboration practices specific to data engineering are missing, as stated in C3. With hard to discover data communities, this challenge leads to a split between data publishers and data consumers, with few data consumers working together to improve the overall data for every project. Contributing to this challenge are missing tools, prescribing a collaborative workflow that a community can follow. In their interview, E2 notes: "I have the feeling that all the kind of GitHub contribution model came with git and GitHub together, and Wikipedia came with a wiki and maybe we need the tool. [...] It will be very hard to define a workflow if you don't have some tool that kind of works." Existing project forges for software are used, but are missing features [4] and do not enforce a fitting workflow.

Overall, standard tools and artifacts to collaborate on are missing for collaborative data engineering (C4). In open-source software development, participants collaborate on clearly defined artifacts like libraries and frameworks, shared as source code. Currently, most artifacts that are created during collaborative data engineering are either metadata or the resulting data itself (see Table 6), which leads to a mismatch between data artifacts and tools that were made for software as E2 describes: "[SQLite] won't work with git, which works well with text files but not very well with binary files." Collaborating on data sets is also not a viable strategy when data needs to be regularly refreshed to stay up-to-date [26] or for domains in which the data is regularly updated. From their work with open transport data, E2 points out the potential of a standard workflow language that can be used with traditional software tools: "Maybe there will be at some point some workflow process language that will be accepted and will be universal enough, and that can be versioned on GitHub, where you can then have your own version and changing one bit of the process or fork it and rerun it. [...] You need code, you need data and you need workflows. And you want to be able to treat each one of the three of them."

From the interviews, we identified additional technical challenges, summarized in Table 7 with the type technical. Data representation, C5, is a challenge for any data-driven project, but more so for collaborative projects. Obvious issues are different syntaxes to represent the same underlying value, like the use of a comma or a dot to separate decimal numbers and the use of different units such as Celsius or Fahrenheit for temperature. Typing of values is a related problem, with

many domain-specific value types, like postal codes, being unclear or lost in data transfer. This is especially problematic for open data because it is often shared in the form of CSV-files that cannot express custom value types. In addition, interview participants also experienced problems with missing standard taxonomies to create a shared understanding of data, for example in new scientific fields that have many research groups working in them.

Practitioners also highlighted that the tools they are using to work with data are inadequate (C6). Simple data wrangling tools work with small data sets, but cannot deal with larger amounts of data. Programming languages are complex and hard to use, especially for subject-matter experts. Likewise, tools to set up data pipelines are frustrating as E2 explains "So much configuration, so much code, and so much things I copy pasted and I don't how it works [...] And this gets very frustrating. If it's in tools that you do configure by a kind of programming. And if it's a graphical one, you feel frustrated because you spend your time clicking and you don't know where." Then, once a data pipeline is set up, there is little tool support to keep the data pipeline, its executions and the resulting data in sync. For their projects, E8 set up custom code to keep track of all pipeline runs and where the resulting data was saved by updating a database. With the need to regularly update data, this led to a large overhead of custom tool development.

Related to those problems, E8 also pointed out how much harder it is to host infrastructure for data projects (C7) than it is for fields with more existing standards, like mobile application development. This includes infrastructure providers for data (e.g., data lakes) but also for backend code like cloud providers. Even for theoretically simple tasks like hosting code that regularly fetches data from a source and loads it into a data sink, no common patterns exist that are used across a data community. Therefore, data engineers write custom solutions that others have to understand before they can collaborate.

The final technical challenge to collaboration are the bad data sources (C8). These issues include problems with the data itself, like no documentation or missing units for values that need to be verified with subject-matter experts, or semi-structured data like text that is hard to use. Challenges with data sources also extend to how they are made available, for example because of flaky infrastructure or old technology. E8 describes their experience working with data from a news publisher: "You basically get access to a FTP servers via really unsecure connections. It's like in the Middle Ages [...]". For data that is regularly updated, like open transport data, an additional challenge lies in the fact that collaboratively fixing errors in a data set and sharing the corrected data is of limited use because the next time an update is released, the same errors will be present if the fixes are not shared and accepted by the data publisher.

Collaborating with data publishers, for example by contributing back fixes for errors, is often not easy. This challenge is part of the social challenges we have extracted from interviews, shown as C9 in Table 7. For the open data practitioners we have interviewed, it is a problem to get into contact with data publishers and if they can, often no one feels responsible. For the larger data community, this is a challenge to collaboration because important participants from the publisher side do not contribute, either with knowledge about the data structure or by accepting feedback from the community. One of the reasons for data publishers not participating in collaboration is a territorial feeling about the data, a sentiment expressed by E2 as follows: "I published the data so I know how it should be and the community is wrong." These feelings are reinforced by fears from data publishers. Depending on the context, data publishers might fear contributors entering bad data (in the case of open data collaborations like OpenStreetMap) or fear users misusing data to harm others, for example by misrepresenting statistical data. Lastly, data publishers that are forced by law to publish open data, often government agencies, fear the loss of potential business value that is then captured by commercial entities or startups.

These unclear use cases for data also create a different challenge, captured in C10, in which data publishers and potential collaborative date engineering projects are unsure how downstream consumers will use data and what requirements they will have. Because of this, collaborative data engineering efforts have no clear way to evaluate if a data processing step is essential to increase data quality. This leads to costly overhead to reach out to potential consumers or to collaboratively define data schemas as a community, as described in subsection 4.2. The increasing popularity of machine learning poses a special challenge in this regard because their data requirements are unique. As an example given in interviews from the mobility domain, machine learning models can cope with errors in training data reasonably well, but if a data set with accessibility information about wheelchair access includes only a few errors it makes for low-quality data for a mobile app that supports disabled users.

Similar to the technical challenge with data representation described as C5, different impressions of data semantics are a social challenge identified as C11. This challenge manifests itself as many viewpoints on data meaning from different data communities. For open science data, it could mean research communities using lacking standard naming for concepts. Naming issues are also described in more mundane examples, like long discussions about what constitutes a shelter at a bus stop. In international data context, simple naming issues like what to call a subway/metro/underground-train can confuse users. Aside from naming issues, political differences can make collaboration a challenge as well, often concerning geographical data like borders. Even outside conflicts, E2 describes a situation where the border between Germany, Switzerland, and Austria is not clearly defined at the Bodensee, leading to complications with the strict requirement for borders to exist in the data schema of OpenStreetMap.

For collaborative open data projects, missing or misaligned incentives pose a challenge. Especially because data engineering can be, as multiple interviewees point out, “boring work”. Publishers of open data are often forced by law to make their data available, which leads to conflicts, as described in C9. Other domains face similar problems. For scientific data projects, researchers are rewarded less for curating and maintaining data and have to focus on publishing instead. For open collaborations by a data community like hackathons, different forms of incentives like certificates of participation or cryptocurrency are mentioned by interview participants. If a hackathon is run by activists, it is hard to provide similar rewards.

The feeling of ownership of an artifact like a software application or a wiki article can be an incentive to collaborate without a monetary reward. But as E6 points out, with data the ownership typically lies with a data publisher and not the community: “In many collaborative contexts, it can be difficult to really have a sense of ownership over a data set that is produced by a government or some kind of external company. It is quite rare that people produce their own data. So it’s rare that people can invite each other to work on data that they really feel a sense of ownership for.”

Lastly, collaborative data engineering projects suffer greatly from missing knowledge (C13). Data projects are challenging because of the need for knowledge about data engineering, software engineering and subject-matter, as identified by three roles in a project group (see subsection 4.2) and the fourth mediator role. Every interview included a discussion of this challenge. Specific examples include a lack of knowledge about data pipeline tools by subject-matter experts or unfamiliarity with data formats. Some projects, like the heavily regulated medical data E9 dealt with, also have challenges with the missing legal knowledge of contributors. Because the skill sets needed to work with data are so distinct, it is often challenging for contributors to correctly assess the level of knowledge of others. From a software developer viewpoint, E8 explains: “Regular people, don’t how any data format works. You even have to start explaining what a key value store is. [...] With non programmers you have to first start on this level, which is difficult.”

Missing knowledge means that participants must rely on learning information either from subject-matter experts or technical members of a community. If these roles are assumed by few people, the danger of overburdening them with too many questions exists.

5 DISCUSSION

Our results show, that collaborative data engineering projects are part of a fragmented ecosystem. The number of stakeholders involved, subject-matter differences and data cultures with their unique viewpoints and standards make it hard to successfully attempt open collaborative projects with data. It appears as if the open data ecosystem is developing similarly to open-source software but lagging years behind.

Increasing open collaborative data engineering projects would have wider implications on data ecosystems. The ability for data users to share and reuse improved data could lower individual costs. In turn, this would raise the quality and availability of data for the whole ecosystem. With easier to use data, individual projects like hackathons would lose less time to data engineering and could invest more into innovative applications of data.

The availability of high quality, open data sets is especially important with the recent increase in machine learning and artificial intelligence projects. Machine learning models need large amounts of machine-readable data for training and evaluation, meaning they are hard to develop outside commercial contexts. Democratizing access to the underlying data can enable more participants to develop their own models and evaluate existing ones critically.

Building on the insights described in section 4, we contribute a set of guidelines for successful collaborative data engineering projects, summarized in Table 8. In accordance with our research questions, we focus on the social systems and challenges that arise from collaborative work and less on the inherent technical challenges of individual data engineering. These guidelines provide a preliminary framework for practitioners to increase the adoption of collaborative data engineering in their projects. For researchers, these guidelines provide a starting point to extend them into a more complete theory of collaborative data engineering.

Following these guidelines, we make concrete recommendations to increase the adoption and success of open collaborative work during data engineering in the context of open data. An overview of these recommendations is shown in Table 9. Similar to the guidelines, these recommendations should be understood as preliminary. Policymakers and open-data enthusiasts that want to increase collaboration in data engineering should take these recommendations into account, but be open to changing their approach with additional insights.

5.1 Guidelines for Open Collaborative Data Engineering Projects

For all attempts at collaborative data engineering, it is essential to take the realities of data issues into consideration and not plan with an idealized view. In most contexts, the data will be distributed over many locations, regularly updated, of varying but often low quality, hard to access and missing metadata (C5, C8 as described in Table 7). Additionally, especially for open data, data publishers are usually hard to reach and have misaligned incentives that make them unlikely to contribute or resolve issues with their data (C9, C12). These challenges have to be considered in the planning of collaborative data engineering projects, for example by focusing on supporting data users to help themselves instead of trying to improve the source data directly.

Based on the roles in collaborative data engineering, summarized in Figure 4, the focus of collaborative data engineering projects should initially be on the project group as its members are part of every social interaction we have identified. While the project manager and mediator role can be flexible, different viewpoints arise from the triangle of data engineer, software developer and subject-matter expert. Any collaborative data engineering project must make sure to be accessible

ID	Guideline	Based on
G1	Plan with data problems like distributed sources, updates, low-quality and limited access to publishers	C5, C8, C9, C12
G2	Make projects accessible to data engineers, software developers and subject-matter experts	Social Systems, C1, C13
G3	Enable collaboration by agreeing on standards, improving project visibility and curating data	C2, C3, C4, C5, C7, C8, C12
G4	Support projects with tools, built specifically for collaborative data engineering	C1, C4, C6

Table 8. Guidelines To Enable Open Collaborative Data Engineering

and support members with these backgrounds. If a project group is lacking one of these essential roles, adding a member that can fill it should become the highest priority (C1).

With a stable project group, open collaboration with a larger data community can be established, allowing participants to share and re-use artifacts and lowering individual costs. To do so, it is essential to agree on how to share intermediate work results and collaborative workflows (C3 and C4). Finding and connecting with other community members will be a challenge (C2) that must be considered. Potential solutions include improving project and user visibility, providing a robust search, or supporting community members with connector or communicator roles.

Finally, projects should be supported with tools that are specifically built for collaborative data engineering (C4, C6). While collaborative data engineering shares many similarities with open-source software development, the reuse of software engineering workflows and tools for collaborative data engineering work might, in fact, be a detriment to experimentation because they are 'good enough' but not ideal. As described earlier, these tools must be accessible not only to software developers but also data engineers and subject-matter experts. Because these tools are mainly created by software developers, care must be taken to include the other viewpoints in their evaluation as well.

5.2 Recommendations to Increase Open Collaborative Data Engineering in Open Data

ID	Recommendation	Based on
R1	Define a standard artifact to collaboratively develop data pipelines	G1, G2, G3
R2	Adapt proven open collaborative workflows for data engineering	G2, G3
R3	Provide a project forge to drive adoption of standards	R1, R2, G4
R4	Develop tools that make running data pipelines and hosting data projects easier	G4
R5	Support the creation of data communities	G3

Table 9. Recommendations to Increase Open Collaborative Data Engineering in Open Data

Shared standards in a community are important for collaborative work. To that end, a standard artifact to collaboratively develop data pipelines should be defined. This artifact can not be the improved data itself because in many domains (for example schedules in open transport data) data sets are regularly updated and re-released, while contacting the data provider is challenging as described in C9 and C12 (see Table 7). If the cleaned data is shared as collaborative artifact without being able to fix errors at the source, every time new data is released, additional effort will be

required to make the same changes again. Instead, the artifact should describe data pipelines that can be re-executed once the data source changes and apply the previously defined transformations and improvements again. Various options to model data pipelines exist. However, they are often commercial, leading to vendor lock-in and slow innovation. Others are GUI tools, like Apache Hop, that make collaboration complicated. An ideal tool to model data pipelines should be text-based and open to initially reuse the mature software engineering tooling, like version control systems and editors, but allow for rapid evolution of an ecosystem of own tools. As discussed for G2, this artifact must also be accessible for all members of a typical project team. Existing data pipeline solutions, based on general-purpose programming languages and frameworks like Apache Flink, can be used by expert software developers but lead to high barriers to participation, especially from subject-matter experts (C1). A potential collaboration artifact would be a domain-specific language to model data pipelines that can reuse existing software engineering tooling and is intuitive for software developers, but can still be understood by data engineers and subject-matter experts due to its reduced scope and domain-specific concepts.

With this text-based artifact, standard collaboration workflows should be adapted for data engineering. Especially proven approaches from open-source development can be a starting point due to the similarity of collaboration artifacts. However, similarly to tool development, the danger of being stuck with good-enough practices like GitHub's pull request model might stifle innovation that is more appropriate for data engineering. Therefore, existing practices should be evaluated and adapted individually, while ensuring they can be followed by all major roles involved in collaborative data engineering. With inspiration from open-source development, it will be especially important to ensure subject-matter experts are able to participate equally, even if they lack experience in software development.

However, the definition of such artifacts and workflows is not only an academic challenge because they can not enable open collaboration without community adoption. Suggesting new standards must therefore be accompanied by well-crafted tools that provide real value to practitioners to have a realistic chance of being used. In contrast to open-source development, data engineering lacks a centralized and highly frequented project forge that would enable collaborative projects to advertise themselves and be discovered. A project forge for collaborative data engineering projects would be useful in three major ways. First, it can standardize the tools and workflows used in the community by providing them in an easily accessible way, together with project hosting [20]. Additionally, a project forge can be a hub for the curation of high-quality data projects, unifying sources spread of many existing data portals. Finally, a project forge can allow community building by improving the visibility of both data projects and users, reducing challenges like C1 and C2. Initially, this can be done by implementing a search feature, but other possibilities like matching users to projects based on skills or interest using AI algorithms could be explored.

In addition to a project forge, a related software implementation should make running data pipelines and hosting data-driven projects easier. An essential requirement for these tools is to keep pipeline models, executions and the resulting data in sync. Without this functionality, collaborative data engineering projects have to invest a large amount of work in building their own solutions, as described by E8 for challenge C6. Making it easier to execute data pipelines, for example in a standardized cloud environment, enables more contributors to participate in their collaborative development because they need less technical equipment and expertise. This is especially relevant as the skill set needed to build and maintain data pipeline infrastructure at scale is distinct from other software development skills required during data engineering. As such, even project team members that can fulfil the software developer role would benefit from the reduced scope of their responsibilities.

Finally, the creation of data communities should be actively supported. Fundamentally, this means creating aligned incentives (in contrast to the missing incentives described in C12) for all participants in open collaborative data engineering ecosystems. As an example, with easier hosting for data projects and a centralized project forge based on the recommended tools, it will be possible to highlight projects that are only possible because of the underlying data. This in turn will reflect positively on public data providers and create incentives for them to improve the data they publish and to engage in the community. Making work visible to create incentives is also possible by supporting connector and communicator roles, as well as increasing the amount of public work (see 8).

6 LIMITATIONS

The search for our systematic literature review was scoped to Google Scholar and Scopus and only included peer-reviewed articles in English or German. The depth of the literature pool could be improved by extending the search to additional languages or including gray literature. However, we supplemented the data from literature with interviews and explicitly confirmed information extracted from articles with practitioners.

The sampling of participants in our qualitative interview study imposes some limitations on the resulting qualitative data. We sampled for theoretical diversity, actively approaching participants that would provide us with new insights instead of building a statistically representative sample of the data community. While we consider this choice appropriate for our goals to describe the diversity of social systems and challenges during open collaborative data engineering, we cannot make statistical inferences from the data.

Data extraction was performed by descriptive data synthesis, both for the results of the literature review and for the qualitative interview study. Ideally, we would use additional qualitative data analysis methods to deepen our understanding of the data and describe the relationships between participants in data projects and challenges in more depth.

Bias is a possible threat to validity because open data is often published by government agencies. For this reason, much of the academic literature and practitioners concern themselves with open government data. As our goal was not to attempt quantitative data synthesis, the thread is less relevant, but the danger of missing information from other data domains remains. To mitigate this, we tried to sample practitioner interviews from other data domains and asked for practitioner feedback (see Table 2) for the results of our academic literature review.

The presented guidelines and recommendations in section 5 are based on insights gained from the literature and interviews. However, they were not independently evaluated. In combination with the non-representative sample of interview participants discussed earlier, these contributions should be understood as preliminary. We took care to point this distinction out and separated their presentation from the results of the systematic literature review and interview study itself.

7 CONCLUSION

In summary, we aimed to answer three research questions related to open collaboration during data engineering by open data users: *Which elements of collaboration systems for data engineering by open data users are described in literature?*, *How and in which roles do participants in collaboration systems for data engineering interact socially?* and *What are challenges to collaboration in data engineering and why?*

We provided an overview of elements in collaboration systems for data engineering by data users by performing a systematic literature review. We find data users from heterogeneous backgrounds that use a variety of tools to process data. The collaborative data engineering processes described in the literature include a wide range of activities, from technical, like writing scripts to fix errors,

to social like finding and asking subject-matter experts for advice. While we could identify and categorize individual activities, we could not find a standard collaborative process that is followed across all data engineering projects.

To describe the social interactions in data projects in more detail, we performed a qualitative interview survey with participants. From the interviews, we extracted descriptive data about roles that contributors fulfil, as well as unique social interactions. The results indicate that the five roles of data engineer, software developer, subject-matter expert, mediator, and project manager make up the core of a collaborative data engineering project (for an overview, see Figure 4). We identified additional roles in the larger data community, as well as supporting roles that contribute special skills when needed. We describe important interactions between the roles that must happen to mediate between the different viewpoints, collaboratively define data schemas or prioritize data. Additionally, we found that the recent trend of public work also applies to collaborative data engineering projects, especially hackathons.

Based on qualitative data both from literature and interviews, we identified challenges to collaborative data engineering. Due to the complex, collaborative work required, these challenges are both technical and social. Technical challenges relate mostly to bad quality data sources, tools that are no good fit for data engineering and the complexities of hosting and maintaining data projects. Social challenges stem from the interactions in the larger data community, especially conflicts and little shared understanding between data publishers, project members and data consumers. For individual collaborative data engineering projects, incentives are often misaligned or missing. Because many specialized skills are needed to work with data, missing technical, subject-matter or legal knowledge is a central challenge that has to be resolved. A summary of the challenges is shown in Table 7.

Building on our insights from the results, we described guidelines to follow to make collaborative data engineering projects successful, such as planning with data problems from the start and ensuring that projects are accessible to the main project group roles of data engineer, software developer and subject-matter expert. We point out the importance of agreeing on standards, making projects discoverable and curating data, as well as providing purpose-built tools instead of just relying on infrastructure built for collaborative software engineering. To increase adoption of open collaborative data engineering in open data, we made concrete recommendations. First, to create a standard collaboration artifact as well as workflows, then building adequate tooling in the form of a project forge and a cloud environment to execute data pipelines to drive their acceptance. Finally, the creation of data communities must be actively supported by creating incentives and enabling connecting social roles. The guidelines and recommendations can be found in Table 8 and Table 9 respectively.

Our contributions are relevant for a variety of audiences. Practitioners that want to introduce or increase collaboration during data engineering should keep the social systems described in Figure 4 in mind, be aware of the challenges described in Table 7 and consider following the guidelines presented in Table 8 to make their project successful. For policymakers and enthusiasts working with open data, our recommendations (see Table 9) to improve the open data ecosystem are relevant as well. Lastly, researchers can make use of the overview of activities, artifacts, tools, and participants (Table 3 - Table 6) and social systems from Figure 4 to understand the domain of open collaborative data engineering better.

More work is needed to deepen our knowledge about the identified challenges and potential solutions. While we focused on diversity, additional research is needed to highlight the differences between data projects, for example between open and closed data, and compare the social systems they create. The guidelines and recommendations we suggested based on the results should be

evaluated and extended by applying them to real open collaborative data engineering projects and observing their impact.

Finally, we would like to extend our work with a larger, quantitative survey among data engineering practitioners to find additional challenges and statistically representative insights.

ACKNOWLEDGMENTS

This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17045 Software Campus 2.0 (Friedrich-Alexander-Universität Erlangen-Nürnberg) as part of the Software Campus project 'JValue-OCDE-Case1'. Responsibility for the content of this publication lies with the authors.

The authors would like to thank the anonymous interview participants for their time and insight, as well as Georg-Daniel Schwarz for his extensive and constructive feedback. Additionally, the detailed feedback from the anonymous reviewers was helpful to improve the scope and clarity of the manuscript.

REFERENCES

[1] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. 2008. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/1453101.1453107

[2] Glenn A Bowen. 2008. Naturalistic inquiry and the saturation concept: a research note. *Qualitative research* 8, 1 (2008), 137–152.

[3] Gemma Catolino, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Filomena Ferrucci. 2020. Refactoring community smells in the wild: the practitioner's field manual. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society* (Seoul, South Korea) *(ICSE-SEIS '20)*. Association for Computing Machinery, New York, NY, USA, 25–34. https://doi.org/10.1145/3377815.3381380

[4] Joohee Choi and Yla Tausczik. 2017. Characteristics of collaboration in the emerging practice of open data analysis. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing* (Portland Oregon USA). ACM, New York, NY, USA. https://doi.org/10.1145/2998181.2998265

[5] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1277–1286. https://doi.org/10.1145/2145204.2145396

[6] Egon G Guba. 1981. Criteria for assessing the trustworthiness of naturalistic inquiries. *Ectj* 29, 2 (June 1981), 75. https://doi.org/10.1007/bf02766777

[7] Philip Heltweg and Dirk Riehle. 2023. Challenges to Open Collaborative Data Engineering. In *Proceedings of the 56th Hawaii International Conference on System Sciences* (Hyatt Regency Maui), Tung X Bui (Ed.). 679–688. https://doi.org/10125/102714

[8] J D Herbsleb, A Mockus, T A Finholt, and R E Grinter. 2001. An empirical study of global software development: distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001.* 81–90. https://doi.org/10.1109/ICSE.2001.919083

[9] Harrie Jansen. 2010. The Logic of Qualitative Survey Research and its Position in the Field of Social Research Methods. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* 11, 2 (2010). https://doi.org/10.17169/fqs-11.2.1450

[10] Miguel Jiménez, Mario Piattini, and Aurora Vizcaíno. 2009. Challenges and Improvements in Distributed Software Development: A Systematic Review. *Advances in engineering software* 2009 (June 2009). https://doi.org/10.1155/2009/710971

[11] Hanna Kallio, Anna-Maija Pietilä, Martin Johnson, and Mari Kangasniemi. 2016. Systematic methodological review: developing a framework for a qualitative semi-structured interview guide. *Journal of advanced nursing* 72, 12 (Dec. 2016), 2954–2965. https://doi.org/10.1111/jan.13031

[12] Barbara Kitchenham. 2004. Procedures for performing systematic reviews. *Keele, UK, Keele University* 33, 2004 (2004), 1–26.

[13] Mikkel B Kjærgaard, Omid Ardakanian, Salvatore Carlucci, Bing Dong, Steven K Firth, Nan Gao, Gesche Margarethe Huebner, Ardeshir Mahdavi, Mohammad Saiedur Rahaman, Flora D Salim, Fisayo Caleb Sangogboye, Jens Hjort

Schwee, Dawid Wolosiuk, and Yimin Zhu. 2020. Current practices and infrastructure for open data based research on occupant-centric design and operation of buildings. *Building and environment* 177 (June 2020). https://doi.org/10.1016/j.buildenv.2020.106848

[14] Thomas D LaToza and André van der Hoek. 2016. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software* 33, 1 (Jan. 2016), 74–80. https://doi.org/10.1109/MS.2016.12

[15] Martin Lnenicka and Jitka Komarkova. 2019. Big and open linked data analytics ecosystem: Theoretical background and essential elements. *Government information quarterly* 36, 1 (Jan. 2019), 129–144. https://doi.org/10.1016/j.giq.2018.11.004

[16] Gustavo Magalhaes, Catarina Roseira, and Sharon Strover. 2013. Open government data intermediaries: a terminology framework. In *Proceedings of the 7th International Conference on Theory and Practice of Electronic Governance* (Seoul, Republic of Korea) *(Icegov '13)*. Association for Computing Machinery, New York, NY, USA, 330–333. https://doi.org/10.1145/2591888.2591947

[17] Kevin O'Leary, Rob Gleasure, Philip O'Reilly, and Joseph Feller. 2020. Reviewing the Contributing Factors and Benefits of Distributed Collaboration. *Communications of the Association for Information Systems* 47, 1 (2020), 24. https://doi.org/10.17705/1CAIS.04722

[18] Arie Purwanto, Anneke Zuiderwijk, and Marijn Janssen. 2020. Citizen engagement with open government data. *International journal of electronic government research* 16, 3 (July 2020), 1–25. https://doi.org/10.4018/ijegr.2020070101

[19] Vijay Janapa Reddi, Greg Diamos, Pete Warden, Peter Mattson, and David Kanter. 2021. Data Engineering for Everyone. *CoRR* abs/2102.11447 (2021). arXiv:2102.11447 https://arxiv.org/abs/2102.11447

[20] Dirk Riehle, John Ellenberger, Tamir Menahem, Boris Mikhailovski, Yuri Natchetoi, Barak Naveh, and Thomas Odenwald. 2009. Open Collaboration within Corporations Using Software Forges. *IEEE Software* 26, 2 (March 2009), 52–58. https://doi.org/10.1109/ms.2009.44

[21] Erna Ruijer and Albert Meijer. 2020. Open government data as an innovation process: Lessons from a living lab experiment. *Public performance & management review* 43, 3 (May 2020), 613–635. https://doi.org/10.1080/15309576.2019.1568884

[22] Micah J Smith, Jürgen Cito, Kelvin Lu, and Kalyan Veeramachaneni. 2021. Enabling Collaborative Data Science Development with the Ballet Framework. *Proc. ACM Hum.-Comput. Interact.* 5, Cscw2 (Oct. 2021), 1–39. https://doi.org/10.1145/3479575

[23] Micah J Smith, Roy Wedge, and Kalyan Veeramachaneni. 2017. FeatureHub: Towards Collaborative Data Science. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 590–600. https://doi.org/10.1109/dsaa.2017.66

[24] Sharon Spall. 1998. Peer Debriefing in Qualitative Research: Emerging Operational Models. *Qualitative inquiry: QI* 4, 2 (June 1998), 280–292. https://doi.org/10.1177/107780049800400208

[25] Damian A Tamburri, Rick Kazman, and Hamed Fahimi. 2023. On the Relationship Between Organizational Structure Patterns and Architecture in Agile Teams. *IEEE Transactions on Software Engineering* 49, 1 (Jan. 2023), 325–347. https://doi.org/10.1109/TSE.2022.3150415

[26] Ignacio G Terrizzano, Peter M Schwarz, Mary Roth, and John E Colino. 2015. Data Wrangling: The Challenging Yourney from the Wild to the Lake.. In *CIDR*. Asilomar.

[27] V A Thurmond. 2001. The point of triangulation. *Journal of nursing scholarship: an official publication of Sigma Theta Tau International Honor Society of Nursing / Sigma Theta Tau* 33, 3 (2001), 253–258. https://doi.org/10.1111/j.1547-5069.2001.00253.x

[28] Dakuo Wang, Justin D Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. 2019. Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI. *Proc. ACM Hum.-Comput. Interact.* 3, Cscw (Nov. 2019), 1–24. https://doi.org/10.1145/3359313

[29] Jin Xu, Yongqin Gao, S Christley, and G Madey. 2005. A topological analysis of the open souce software development community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* (Big Island, HI, USA). IEEE, 198a–198a. https://doi.org/10.1109/hicss.2005.57

[30] Amy X Zhang, Michael Muller, and Dakuo Wang. 2020. How do Data Science Workers Collaborate? Roles, Workflows, and Tools. *Proc. ACM Hum.-Comput. Interact.* 4, Cscw1 (May 2020), 1–23. https://doi.org/10.1145/3392826

[31] Anneke Zuiderwijk, Marijn Janssen, and Chris Davis. 2014. Innovation with open data: Essential elements of open data ecosystems. *Information polity* 19, 1,2 (June 2014), 17–33. https://doi.org/10.3233/ip-140329

A INTERVIEW DOCUMENTS

A.1 Interview Handout

Challenges to Open Collaborative Data Engineering - Interview Handout

Research Context

This interview is part of a research project about collaboration during data engineering. When collaborating, multiple people work together to achieve a common goal. During data engineering, raw data is made available for further use. Examples are adding structure, fixing errors or writing documentation. Our goal is to help data engineers collaborate, ultimately providing access to higher quality data for all participants.

We are interested in interviewing people that have attempted or contributed to a data-driven project (e.g., a written report, software application, or data collection) and collaborated with others to process data for use. The interview will include questions about the people and tools you worked with, your experience with data engineering and challenges you might have encountered.

If you have a question or concern, please contact Philip Heltweg (philip.heltweg@fau.de) at the Professorship for Open-Source Software, Friedrich-Alexander-Universität Erlangen-Nürnberg.

Interview Process

These steps are part of the interview process and follow up:

1. The researcher provides this interview handout to set expectations and context for the interview
2. We decide on a date, time and software for the interview
3. During the interview
    a. We start with an informal conversation that is not recorded
    b. After clearly stating that recording starts, we start the official interview
    c. After clearly stating that recording stops, we finish the conversation
4. The researcher transcribes and pseudonymises the recording
5. The researcher shares the pseudonymised transcription with you
6. After your agreement, the data can be used in upcoming publications related to the research project

Data Use

During the interview, we collect raw audio- and videodata. From that data, we create a pseudonymized transcript, using automatic speech recognition services and manual verification. Information about the pseudonyms is stored in a separate location and never combined with the transcripts.

After your agreement, we will use the pseudonymized interview transcript for our research project. To do so, the transcript might be part of qualitative data analysis and be partially quoted or published in full as part of an academic publication. We will not share information related to the pseudonyms used or use your data outside of the context of the research project.

A.2 Interview Guide

Interview Guide

Before the interview

- Remind yourself to be mindful of the participants time, mention end of interview time
- Give a short introduction about the context of the research
    - Define data engineering, collaboration, open collaboration
        - During data engineering, raw data is made available for further use. Examples are adding structure, fixing errors or writing documentation.
        - When collaborating, multiple people work together to achieve a common goal.
        - People are openly collaborating, if outside people can find and join the project, if decisions are made based on agreement rather than dictated by hierarchy, and if people can choose their own processes and work tasks in agreement with others. Examples of open collaboration are open-source software development or content authoring in wikis.
- State the themes / structure of the upcoming interview

During the interview

- Theme: Demographic data
    - Company, job title, role
    - For past data engineering projects, typical: Data domain, role (hobbyist vs. professional), Country, open or closed data
- Theme: Collaborative data engineering
    - Do you think it is possible to make data engineering its own activity, separate from data analytics or machine learning?
    - Do you think data engineering can be split into a generic part that is useful to many projects and a project-specific part?
    - Do you think this generic part of data engineering can be done in collaboration with others?
- Theme: Collaboration systems in data engineering
    - Who have you collaborated with before during data engineering? What are their roles?
    - How and when did you interact with them?
    - What collaboration and data-engineering tools have you used before?
- Theme: Challenges to open collaborative data engineering
    - Which social or cultural challenges have you faced during collaborative data engineering?
        - Workflows (Is there a standard? Can they be improved for data engineering?)
        - Domain specific challenges?
    - Which technical challenges have you faced during collaborative data engineering?
        - Programming languages (Is there a standard? Can they be improved for data engineering, e.g. to work with domain experts?)
        - Collaboration tools like GitHub (Is there a standard? Can they be improved for data engineering, e.g. to work with domain experts?)
    - Have you encountered these previously identified challenges?
        - Need for specialized skills but high barriers to participation
        - Finding and connecting with other community members
        - No standard tools or artifacts
        - No well-understood collaboration practices
- Have we missed any question about collaboration during data engineering that we should have asked?

After the interview

- Disable Zoom recording
- Follow up with transcript

C

Paper 3: An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures

In this appendix, the paper *An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures.* is reproduced in full. The article was originally published as:

Heltweg, P., Schwarz, G.-D., Dirk, R., & Felix, Q. (2025). An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures. Software: Practice & Experience. https://doi.org/10.1002/SPE.3409 [20]

[1] *https://creativecommons.org/licenses/by/4.0/*

Software: Practice and Experience

WILEY

RESEARCH ARTICLE OPEN ACCESS

An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures

Philip Heltweg | Georg-Daniel Schwarz | Dirk Riehle | Felix Quast

Professorship for Open-Source Software, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

Correspondence: Philip Heltweg (philip@heltweg.org)

Received: 22 July 2024 | **Revised:** 11 November 2024 | **Accepted:** 6 January 2025

Funding: This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17045 Software Campus 2.0 (Friedrich-Alexander-Universität Erlangen-Nürnberg) as part of the Software Campus project 'JValue-OCDE-Case1'.

Keywords: data engineering | domain-specific language | empirical study | evaluation | open data | programming language

ABSTRACT

A large part of data science projects is spent on data engineering. Especially in open data contexts, data quality issues are prevalent and are often tackled by non-professional programmers. We introduce and evaluate Jayvee, a domain-specific language for data engineering aimed at reducing barriers to building data pipelines. We show that a structured DSL can have positive effects on speed, ease of use, and quality for data engineering by non-professional developers. For this, we present an empirical quantitative study, in which we compare the performance of students as proxies for non-professional programmers using Jayvee with Python and Pandas. We search for reasons for the empirical findings using a follow-up interview study on how using a DSL changes how non-professional programmers build data pipelines. Participants solve a subset of tasks faster, more easily, and with higher quality when using Jayvee compared to Python. Interviewees describe tradeoffs regarding the DSL's more limited features, stricter code structure, and explicit descriptions. Jayvee is found to be more approachable, which leads to a more guided development flow. New data engineering languages should provide good tooling and documentation, plan how to visualize intermediate data and consider new development workflows involving tools like ChatGPT to find adoption.

1 | Introduction

Data is the foundation for many innovative apps and, increasingly, AI applications. To be usable, data must be available in a format that fits the application and is of high quality. Data engineering, the activity of making data accessible, reliable, and useful for later use, is a large part of any data science project.

This additional work is not only a challenge to the usefulness of large collections of closed data, for example, in internal data warehouses [1], but especially for open data—a source of large amounts of theoretically usable data with an existing ecosystem of data publishers, intermediaries, and users [2].

In addition to technical challenges, the expertise of human subject-matter experts is often required to make complex data sets available for further use [3]. However, general-purpose programming languages (GPLs) with libraries focused on data engineering are complicated and have a steep learning curve for non-professional programmers. Additionally, they are non-trivial to set up and operate, especially when dealing with large amounts of data.

Instead, various visual programming tools have been suggested as alternatives with a lower technical barrier to entry. While easier to use than GPLs, these tools often use proprietary formats that cannot make use of existing text-based solutions, like line-based diffing of different versions of a model. As a result, they are difficult to apply in larger projects and maintain long-term.

In summary, current solutions are either optimized for professional software engineers who implement data pipelines with complex GPLs or for subject-matter experts who work with limited visual programming tools.

A potential middle-ground between GPLs and visual programming tools is domain-specific languages (DSL). A text-based DSL for building data pipelines could reduce complexity and allow subject-matter experts to apply their existing experience, while still allowing the reuse of existing software engineering infrastructure like integrated development environments.

An important distinction for text-based DSLs can be made between internal DSLs that extend an existing host language and external DSLs that are separate languages that require their own tooling but provide the most flexibility [4]. Examples of internal DSLs can include domain-specific frameworks such as Rails for Ruby, while an example of a well-known external DSL is SQL. In the context of this study, we use the term DSL to refer to external DSLs that do not rely on a host language.

The overarching goal of our research is to explore whether an external DSL can achieve a sweet spot for data engineering by subject-matter experts and if so, which implementation decisions are the best. This study makes the first step in this overarching theme to explore how using a DSL affects data engineering, mainly by collecting qualitative data from users. For this, we initially worked intentionally broad, with a focus on qualitative data from users to build an initial theory of how using a DSL affects data engineering. Based on the insights from this exploratory work, we can generate hypotheses to iteratively improve our understanding of what makes DSLs work best for data engineering. In future work, we will test the impact of specific features with controlled experiments.

In this article, as a first step, we explore if a DSL can be a viable alternative to a GPL, and what effects the use of a DSL has on the development process and the quality of the final results. Using a mixed methods research design, consisting of descriptive surveys followed by a qualitative interview study, we answer the following research questions:

Research Question 1: *Is using a DSL for data engineering a viable alternative to a GPL with a data engineering library?*

Research Question 2: *What is the user's perception of difficulty and quality of results using a DSL compared to a GPL with libraries?*

Research Question 3: *What are the effects of using a DSL for data engineering compared to a GPL with libraries?*

With our research, we make the following contributions:

- We showcase the feasibility of using a DSL for data engineering with non-professional programmers.
- We evaluate to what extent non-professional programmers can use a DSL for data engineering.
- We describe the main effects of using a DSL over a GPL with libraries for data engineering.
- We highlight important challenges for developing new languages for data engineering that should be considered in future implementation efforts.

2 | Related Work

An adjacent research field to data engineering with open data is research into scientific workflows and associated workflow systems that orchestrate independent scientific tools into data analysis workflows [5]. Scientific workflows have specific requirements, such as high reproducibility or infrastructure independence. While Jayvee, the language we introduce and evaluate in this study, could be used as a tool in a scientific workflow, we evaluate the effects of using a DSL for data engineering in a more general setting of improving data sets of any complexity, often from open data sources, for downstream use and do not cover later steps in the data science process such as data analysis.

One of the many tools used to define scientific workflows is the Common Workflow Language (CWL) [6]. The CWL allows scientists to define portable workflows of command-line-based tools based on container technologies for data analysis. However, the CWL explicitly mentions workflows that interact with stateful web services or need scheduling as being out of scope, requirements that are common to access open data from data portals or update data when sources change (such as transport schedules in mobility data). We, therefore, consider Jayvee and the evaluation of its effects as complementary work with a slightly different focus on open data sources. The empirical insights on the effects of using a DSL for data engineering will be applicable to other workflow specification languages as well.

During data engineering on open data, practitioners mostly rely on adequate but not well-adapted tools from software engineering [7]. However, several software artifacts that aim to support data engineering have been suggested and empirically evaluated.

Liu et al. evaluate Governor, a tool to provide DBMS capabilities to open data portals [8]. Their goal is to support end users without technical skills (such as journalists) with search, data understanding, and integration of open data. Users could work efficiently with the tool, but were missing more data transformation functions. We consider our work complementary because a DSL could provide more complex data engineering functionality while still lowering technical barriers to data engineering.

Data identification, data understanding, and relationship discovery are identified as important problems in data engineering by Bogatu et al., who present and evaluate Voyager, a tool to support data scientists in these tasks, with results showing considerable time reductions [9]. In contrast to Jayvee, Voyager uses algorithmic insights into the underlying data and does not aim to enable manual work by human experts.

General-purpose languages (GPLs), like Java or C++, enable programmers to develop applications in any domain. In contrast, DSLs are less generally applicable but more expressive in the limited domain they cover. Benefits include increased productivity, lower maintenance expenses, and enabling a larger pool of contributors compared to GPLs [10]. Widely adopted DSLs are, for example, HTML for the domain of hypertext web pages, LaTeX for the domain of typesetting, or SQL for the domain of database queries.

Kosar et al. conducted a systematic mapping study in 2016 to report on the state of the research field of domain-specific languages [11]. While most studies focus on the domain analysis, design, and implementation of DSLs, studies on validation and maintenance are rare. do Nascimento et al. performed a systematic mapping study in 2012 and found that only approximately a third of the investigated studies include evaluation and validation research [12] such as ours.

This study is an empirical evaluation of a DSL in the data engineering domain, going beyond the general evaluation of a DSL against its requirements. Kolovos et al. list important requirements for DSLs, like simplicity and quality, which we focus on in this study [13].

There is a stream of existing evaluations of DSLs in multiple domains. For example, Meliá et al. compare text-based versus graphical notations in the domain of solving software maintenance tasks [14]. In their context, the textual notation won in terms of efficiency and preference of the participating students. Instead, we evaluate a textual DSL against a textual GPL. Kosar et al. compare a DSL with an application library in an experiment with 36 programmers in the domain of graphical user interface construction [15]. Their findings reveal that XAML (the DSL) performs significantly better than C# forms (the GPL) regarding program understanding in all cognitive dimensions. Johanson and Hasselbring evaluate a DSL for ecosystem simulation specifications as a candidate for a non-technical domain [16]. They report increased correctness and reduced time spent per task.

In the domain of model-driven engineering, dedicated model transformation languages (MTLs) are studied that allow the generation of multiple artifacts, such as source code or different views from one model. Höppner et al. conducted an empirical study with semi-structured interviews among 56 experienced researchers and practitioners in the field of model transformations on factors influencing the properties of MTLs [17]. Their results show that one of the largest barriers to the adoption of MTLs is the quality of tooling, an experience that is mirrored by our data as well. Our study adds additional empirical data on the effects of external DSLs from a different domain (data engineering instead of model transformation) and population (novice developers instead of experienced practitioners).

Other domains in which DSLs have recently been evaluated include traffic simulation and type inference rules. In both cases, the DSL was compared with an appropriate GPL with libraries using a controlled experiment, with results showing improved efficiency when working with a DSL. In their work, Hoffmann et al. evaluate the DSL Athos compared to JSpirit, a library for Java [18] for work by subject-matter experts. Klanten et al. describe a controlled experiment comparing the readability of type inference rules in a DSL with Java [19] The authors also describe that empirical studies are rare in the field of programming language design. Similarly to these studies, we contribute additional data in the domain of data engineering to reduce the lack of empirical findings in the field.

Alongside evaluations of single DSLs, some meta-studies include evaluations of multiple DSLs. Kosar et al. compare a family of three controlled experiments in three domains: feature diagrams, graph descriptions, and graphical user interfaces [20], also with student participants. In terms of comprehension correctness and comprehension efficiency, the DSL performed significantly better than the GPL in all three settings. A later replication study confirmed these results, while allowing the use of an IDE to make the experiment setting more realistic [21]. This study strengthens the findings of these overarching ones by providing another DSL evaluation in the domain of data engineering, which, to the best of our knowledge, has not been consulted yet for such a comparison. The data engineering domain might be especially interesting, as the borders between non-programmers engaging in data engineering activities and software developers are fluent. This introduces special challenges like the need to collaborate on a shared artifact with vastly different viewpoints and experience levels with software development.

3 | Jayvee Examples

Our research goal is to test whether an external DSL is better than using a GPL with libraries for data engineering. To this end, we chose to implement a DSL that does not extend a host language to be able to test our hypotheses and collect qualitative data.

Because it is domain-specific, programs in the DSL can be structured according to the pipes and filters architecture [22, 23]. These programs can be represented as directed graphs, making them a good basis for visual programming tools. Their structure aligns naturally with the visualization of pipelines by boxes and arrows and the mental model that non-professional programmers use to reason about data pipelines.

We implemented a domain-specific language called Jayvee to model data pipelines, structured with pipes and filters as first-class programming constructs. The project is available as open source under the AGPL-3.0-only license on GitHub[1]. The language itself is implemented as an external DSL, based on a context-free grammar using the Langium[2]grammar language. Langium provides TypeScript representations of the semantic model of Jayvee and a parser to instantiate an abstract syntax tree (AST) from Jayvee models. Because Jayvee can not re-use tooling of a host language, we have additionally implemented a language server using the language server protocol and a VSCode extension based on it. Jayvee's execution semantics are defined by a reference interpreter implementation based on the generated AST interfaces.

Jayvee aligns as closely as possible with the mental model of data pipelines as a directed acyclical graph of connected processing

steps, similar to the well-known pipes and filters architectural pattern used for data processing.

Thus, Jayvee defines the following core concepts, each marked with a keyword in the language:

Blocks (keyword `block`): Blocks are the building blocks of Jayvee, and each represents a processing step on the data. In the pipes-and-filters pattern, those blocks are the filters. We chose the term "block" because we felt the term filter would not represent the breadth of the intended computational work. Each block can be referenced from other language elements by a user-provided name. The behavior of a block is specified by the block's type, which refers to a built-in element after the `oftype` keyword. The body of the block, wrapped in curly braces, allows users to further configure the block's behavior by assigning values to properties, depending on the block type. For example, the `CarDataCSVExtractor` in Listing 1 defines an extractor block for HTTP data that downloads a file from a given URL. All available block types are listed in Jayvee's documentation[3].

Pipes (syntax: `->`): Pipes are connectors between blocks and indicate a sequential data flow from the first to the second block, both referenced by name. Instead of defining pipes on only pairs of blocks, users can also define chains of pipes that link a sequence of blocks with an arbitrary length.

Pipelines (keyword `pipeline`): Pipelines are the central abstraction element, bracketing blocks and pipes, each containing a sequence of pipes between blocks in its body (indicated by curly braces). Such a sequence of pipes describes the data flow from source blocks (without an input) through downstream transformation blocks (with inputs and outputs) until it exits the pipeline in a sink block (without an output). Pipelines can contain block definitions, but blocks can also be defined outside a pipeline.

```
pipeline CarDataPipeline {
  CarDataCSVExtractor
    -> CarDataInterpreter
    -> CarDataSQLiteLoader;

  block CarDataCSVExtractor oftype CSVExtractor
    url: "https://example.org/data.csv";
    enclosing: '"';
  }
  block CarDataInterpreter oftype TableInterpreter {
    header: true;
    columns: [
      "name" oftype text,
       // ... Further value type assignments
    ];
  }
  block CarDataSQLiteLoader oftype SQLiteLoader {
    table: "Cars";
    file: "./cars.db";
  }
}
```

LISTING 1: Example pipeline structure definition in Jayvee

Listing 1 gives an example of a minimal pipeline. The presented pipeline extracts a CSV file about cars from an HTTP source, assigns value types to its columns, and loads it into an SQLite database. By syntactically separating the definition of the pipeline structure (in Listing 1, lines 2–4) from the details of property assignments in blocks, Jayvee provides a high-level overview of every step that is executed in a pipeline.

In comparison to Python with libraries such as Pandas, the explicitly modeled blocks and pipes lead to models with a consistently enforced structure and a clear order of steps.

Consider one of the ways that users could choose to download a GTFS file and extract data about stops from it in Python, shown in Listing 2.

```
import pandas as pd
import urllib.request
from zipfile import ZipFile

urllib.request.urlretrieve
("https://example.org/GTFS.zip", "data.zip")
ZipFile("data.zip").extract("stops.txt")

df = pd.read_csv("stops.txt")
// ... Further processing
```

LISTING 2: Downloading and accessing stops in a GTFS data set using Python

A roughly equivalent Jayvee pipeline is shown in Listing 3. Note how an overview of the pipeline content and order is provided by lines 2–4, before a reader looks at further details of the blocks.

```
pipeline StopsPipeline {
  GTFSFeedExtractor
    -> StopsFilePicker
    -> StopsCSVInterpreter
    // ... Further processing

  block GTFSFeedExtractor oftype GTFSExtractor {
    url: "https://example.org/GTFS.zip";
  }
  block StopsFilePicker oftype FilePicker {
    path: "/stops.txt";
  }
  block StopsCSVInterpreter oftype CSVFileInterpreter {
    enclosing: '"';
  }
}
```

LISTING 3: Downloading and accessing stops in a GTFS data set using Jayvee

Additional concepts realized in Jayvee include user-defined value types to filter and validate data and data transformations based on a limited expression language. Please refer to the Jayvee documentation[4]for a detailed overview.

4 | Research Design

We chose a mixed methods approach [24] to answer our research questions on whether using a DSL for data engineering is a viable alternative to a GPL with a data engineering library (RQ1), how the user's perception of difficulty and quality of the results differs between them (RQ2) and what effects the use of a DSL for data engineering has compared to a GPL (RQ3). We planned to first

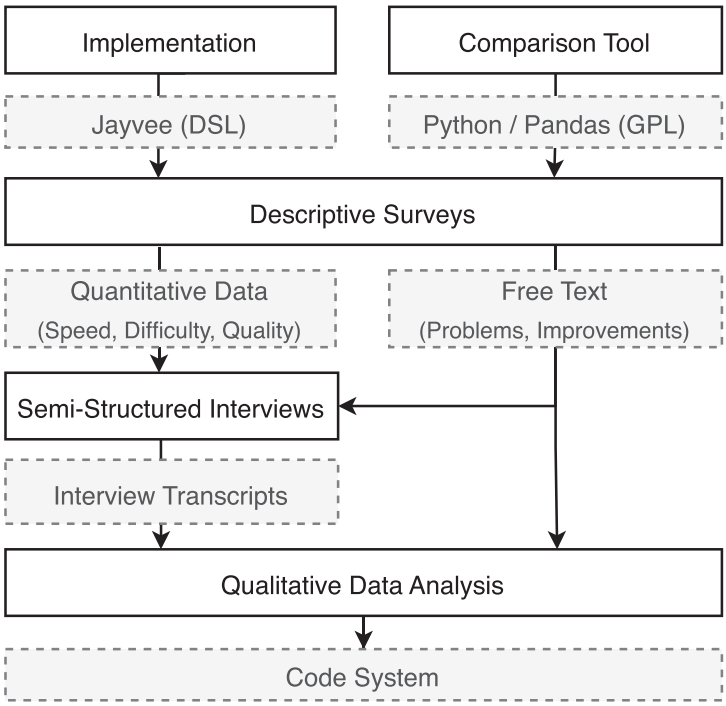


FIGURE 1 | Overview of the research design.

quantitatively test hypotheses and then follow up with qualitative interviews to suggest causal connections.

This research design is well-suited to the exploratory nature of this work, providing an initial insight into the effects of using a domain-specific language that can be extended with follow-up experiments. By employing different research methods, the weaknesses of individual methods can be mitigated, and a more complete picture of the impact of DSLs on data engineering work can be attained.

We consider students taking this course a good proxy for open data practitioners and therefore chose to base the initial empirical study of Jayvee on them [25]. Similar to the students, open data practitioners often have basic experience in programming but come from a wide range of backgrounds, from hobbyists, over statisticians to subject-matter experts [3].

This study was completed as part of a university course on advanced methods of data engineering, mainly taken by master's students who study data engineering, AI, or computer science, over two semesters. The course included five data engineering tasks based on real open data sets, using Jayvee and Python.

Initially, we gathered quantitative data after each task, using a descriptive survey to answer RQ1 and RQ2. Based on the survey insights, we extended and verified the results with interviews after the course had concluded and participants had finished all tasks. This incremental design allowed us to have very focused interviews, answering causality questions that arose from the surveys, and describing the effects of using a DSL over a GPL with libraries to answer RQ3. We employ data and investigator triangulation by gathering quantitative and qualitative data and analyzing it with multiple researchers, as well as presenting our results in peer debriefing sessions to make our results more robust [26, 27]. An overview of the research design is shown in Figure 1.

At the start of each semester, we measured every student's general programming experience and previous experience with Python and Jayvee using a required online questionnaire with previously validated questions according to [28].

Jayvee was introduced with one lecture, and students were provided with the language documentation. During the semester, students solved five graded exercises based on real data sets from the German national access point for transport data, the Mobilithek[5]. The exercises revolved around building ETL pipelines that extract data from an online source, potentially transform it, and load it into a local file sink. The tasks became more difficult over time, introducing students gradually to the domain of data engineering.

The largest amount of open data is provided in tabular data formats, such as CSV or XLS [29]. Available datasets are often small, with the vast majority being under 10 MB in size [30]. Challenges when improving these datasets include the inability to contact data publishers to correct mistakes and regular releases of updated datasets like transport schedules, making one-time data engineering directly changing downloaded datasets less useful. Instead, data users must implement their own error-correcting code and ideally be able to rerun it on updated data sources regularly [3]. Accordingly, the designed exercises were based on real open data sets and targeted the niche of one-time batch processing of tabular data, aligned with the current focus of the DSL. While this use case does not capture the complete domain of data engineering, it is representative of a large percentage of challenges in open data contexts.

TABLE 1 | Task summaries.

No	Task summary
1	Extract a CSV dataset from an HTTP source and assign fitting data types to each column. Save the data to a SQLite database.
2	Extract a CSV dataset from an HTTP source. Transform data shape. Validate data, as defined by categories, integer ranges, and regex patterns. Remove all rows that contain invalid values. Choose fitting data types and save the data to SQLite.
3	Extract a CSV dataset from an HTTP source. Fix invalid format due to included metadata. Handle uncommon encoding to preserve German umlaut characters. Transform data shape by dropping multiple, not adjacent columns. Validate data, handling a special value type of numeric data with leading zeros. Remove any rows containing invalid data. Choose fitting data types and save the data to SQLite.
4	Extract a ZIP file from an HTTP source. Pick one CSV file from the multiple files in the source. Transform the data shape by renaming and dropping columns. Transform data values from Celsius to Fahrenheit. Choose fitting data types and save the data to SQLite.
5	Extract a GTFS file (an open format for transit data in one ZIP file with multiple CSV files) from an HTTP source. Pick one file from the archive. Transform the data shape by dropping columns. Filter the data to only keep rows related to one ID. Validate data values according to integer ranges and keep German umlaut characters intact. Choose fitting data types and save the data to SQLite.

A summary of the tasks is provided in Table 1. The exact exercise descriptions can be found in the accompanying data release.

Students were randomly assigned to two groups of equal size and alternated the language they had to solve each exercise in between Python 3.11 (with pandas 1.5.3) and Jayvee versions 0.0.15, 0.0.16, 0.1.0, and 0.2.0. While the used version of Jayvee changed several times, only a few syntax changes were made, so the usability of the language stayed consistent for the students.

After each exercise, we gathered qualitative and quantitative data using a descriptive online survey developed according to Kitchenham and Pfleeger [31] ("Descriptive Surveys" in Figure 1). The surveys were clearly communicated as optional, with no effect on grades, and included an explicit opt-in to allow the use for publication purposes. The survey software was configured to anonymize all responses, which was also visible to participants.

The questionnaire contained quantitative questions about time spent (*"How many hours did you spend to solve the exercise?"*, numeric), impressions of difficulty (*"How difficult was it to solve the exercise using your programming language?"*, 5-point Likert scale) and quality of the data pipeline (*"How would you rate the quality of the resulting data pipeline?"*, 5-point Likert scale). Additionally, we gathered qualitative data in preparation for the follow-up interview study by asking about problems (*"What problems with the programming language did you encounter during this exercise?"*, free text) and suggestions for improvements (*"What language features or libraries would have made solving the exercise easier?"*, free text). The full survey for exercise 1 can be found in the data release. All other surveys followed the same pattern.

Based on statistical analysis of quantitative survey data, we designed a qualitative survey with semi-structured interviews according to Jansen [32] (step "Semi-structured Interviews" in Figure 1) to better understand the effects of using a DSL instead of a GPL with libraries and perceptions of difficulty and quality of results. We employed convenience sampling, interviewing all students who volunteered for an interview after the semester concluded and grades were already announced. Students were informed about the interview context, process, and questions with a letter to participants beforehand.

The interviews were semi-structured [33] with the main topics being ease of use, quality of results, and challenges as experienced depending on the participant's use of Jayvee or Python. Every interview was concluded with an open-topic question to give participants space to include any insight they considered important. Interviews were performed by two of the authors independently, based on a shared interview guide. After each interview, the audio was transcribed using local software and manually refined to ensure the text was correct. The full letter to participants and the interview guide, including all questions, can be found in the data release.

In a final step, all qualitative data (free text fields from the online surveys and interview transcripts) was analyzed using inductive thematic analysis according to Braun and Clarke [34] ("Qualitative Data Analysis" in Figure 1). First, we familiarized ourselves with the data by reading the primary material actively and noting the first coding ideas. Then, we generated the initial codes by annotating data segments with preliminary names. After open coding, we searched for themes by considering how codes can be combined in different ways to depict a cohesive feature of the data. After the first iteration, we reviewed the themes to clearly distinguish between them and refactor ambiguous ones. Finally, we defined and named the final themes based on their content. The software MaxQDA[6] supported our coding and theme-building process to ensure the traceability of themes and codes back to their origin.

To ensure quality, we regularly requested feedback about elements of the study from other researchers, as summarized in Table 2. We mainly utilized peer debriefings [27] with authors and other researchers who were familiar with the methods used or the domain of data engineering. Furthermore, we presented intermediate results at an internal PhD summit to two research groups consisting of researchers mainly working in the field of software engineering.

TABLE 2 | Feedback methods used during the study.

	Method	Participants	Topic
#1	Peer debriefing	2 researchers	Post-exercise survey
#2	Peer debriefing	3 researchers	Interview guide
#3	Presentation	2 research groups	Intermediate results
#4	Peer debriefing	3 researchers	Open coding of interviews
#5	Peer debriefing	2 researchers	Themes from interviews

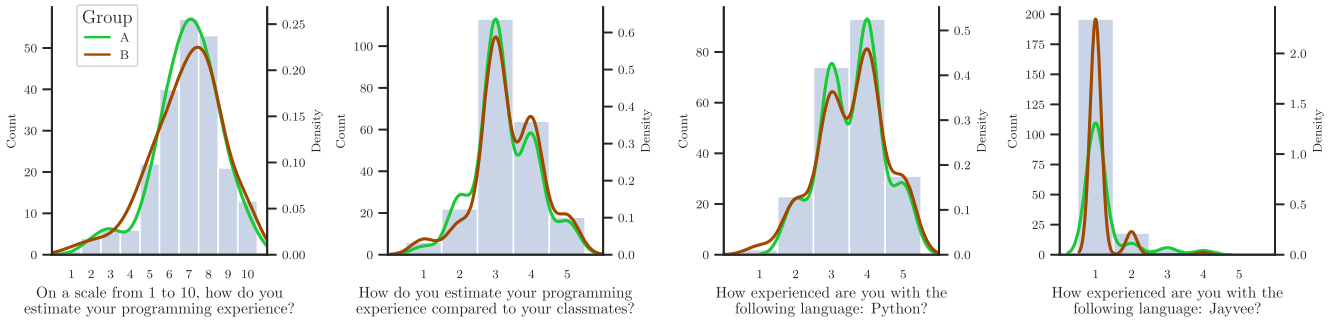


FIGURE 2 | Results regarding student's previous experience from the course entry survey.

TABLE 3 | Sample size, median and Mann–Whitney U and p-value for previous population experience for H_A^{Exp}.

Experience	n_1, n_2	Mdn_1, Mdn_2	U	p (two-sided)
Programming	110, 113	7, 7	6224.0	0.986
Python	110, 113	4, 4	6198.0	0.971
Jayvee	110, 113	1, 1	6547.5	0.223
Jayvee vs. Python	223, 223	4, 1	48398.0	**1.406e-74***

*$p \leq 0.05$.

5 | Results

5.1 | Descriptive Surveys

5.1.1 | Population Description

We gathered quantitative data about previous experience and participants' impressions of time needed, ease of use, and quality of results while solving five exercises alternating between Python with libraries and Jayvee using online surveys as described in Section 4.

We chose students from a course on data engineering because we consider them good proxies for practitioners working with open data. The population consisted of 223 students, mainly in master's studies in computer science, data science, and artificial intelligence, of which 208 completed the course. Their responses to the course entry survey are shown in Figure 2. In addition to the histogram, the kernel-density plots show the distributions of experience in the different groups. Kernel-density plots were chosen as visualization to make it easier to see non-normality, as recommended by [35]. Median programming experience was 7 (of 10), median comparison to classmates 3, and median experience in Python and Jayvee at 4 and 1 (all of 5), respectively.

5.1.2 | Previous Experience

We evaluated whether there were statistically significant differences in previous experience between groups. For the statistical analysis, we used pingouin 0.5.4 [36].

We tested the response distributions for normality using the Shapiro-Wilk test [37] and verified that all were non-normal at $\alpha = 0.05$. Accordingly, we chose the non-parametric Mann–Whitney U test because it is appropriate for the ordinal data of the response options [38, 39]. We decided on the standard significance level of $\alpha = 0.05$.

To ensure previous experience is no confounding factor regarding performance on the tasks, we tested that no statistically significant difference exists between groups regarding previous experience in programming, Python, or Jayvee. We also compared previous experience in Jayvee with previous experience in Python across all students. Table 3 summarizes the results. To detect any difference, we chose a two-sided test with the alternate hypothesis:

H_A^{Exp}: *There exists a significant difference between the previous experience.*

Based on the data, there is no statistically significant difference between groups regarding previous programming, Python, or Jayvee experience. Both student groups were significantly more experienced in Python than in Jayvee. From the visualizations in Figure 2, it is clear that students have much more previous experience with Python than Jayvee (as is expected because Jayvee is introduced as a new language).

5.1.3 | Impressions of Speed, Difficulty, and Quality

After every exercise, we gathered student impressions on the speed, difficulty, and quality of the resulting pipeline, as described in Section 4. Individual response rates for each of the five surveys were 95 responses (42.6%), 61 (27.35%), 25 (11.21%), 35 (15.7%), and 33 (14.8%).

We report an overview of the responses for every dimension of speed, difficulty, and quality and test for statistically significant differences individually for each exercise at a significance level of $\alpha = 0.05$. We removed 15 outlier responses to time according to the standard 1.5 times IQR method.

The Mann–Whitney U test [38] was used because the data is largely non-normal and ordinal. With the smaller sample size for individual exercises, the reduced power of non-parametric tests is a concern. As we are interested in finding out if the use of Jayvee has positive effects on speed, difficulty, and quality compared to Python, we chose one-sided tests to increase the chance of detecting statistically significant effects.

Regarding speed, the alternative hypothesis is:

H_A^{Speed}: *The time needed to solve the exercise is significantly lower using Jayvee compared to using Python*.

Responses are shown in Figure 3, as with the course entry survey, we used kernel-density plots as recommended by [35] to show the distribution of time needed to complete the exercises. More detailed data for each exercise are shown in Table 4.

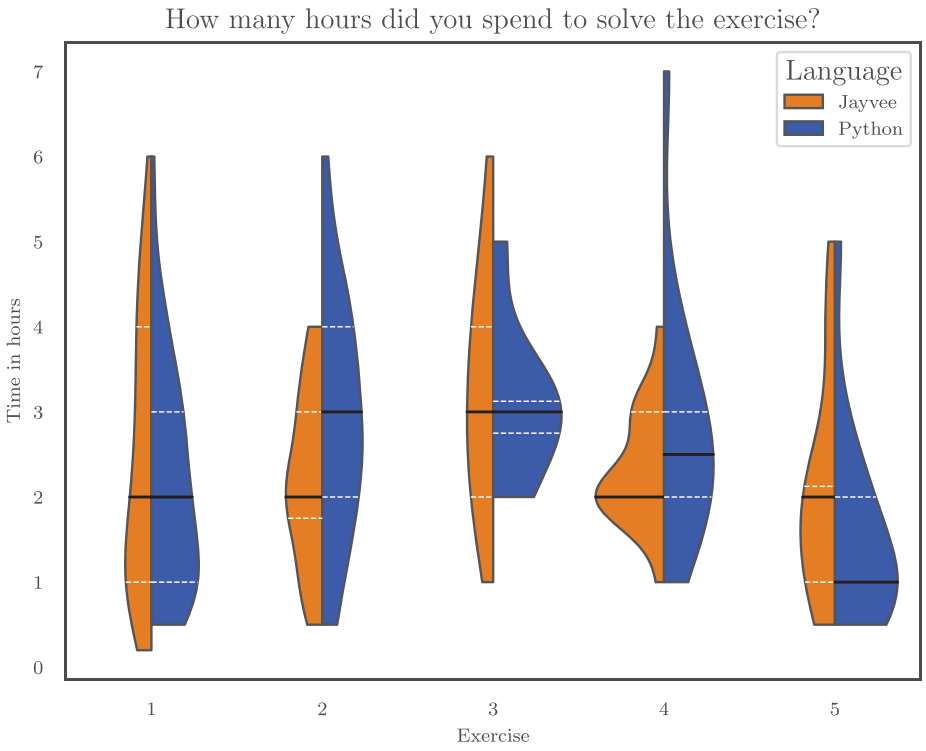


FIGURE 3 | Distribution of time spent per exercise, depending on the language used (lower is better). White bars represent Q_1 and Q_3, the black bar denotes Q_2.

TABLE 4 | Sample size, median and Mann–Whitney U and p-value for time spent on exercises for H_A^{Speed}.

Exercise	n_{jv}, n_{py}	Mdn_{jv}, Mdn_{py}	U	p (less)
Ex1	45, 47	2.0, 2.0	1159.5	0.794
Ex2	28, 24	2.0, 3.0	243.0	**0.042***
Ex3	17, 8	3.0, 3.0	72.0	0.606
Ex4	18, 15	2.0, 2.5	113.0	0.202
Ex5	16, 16	2.0, 1.0	162.5	0.915

*$p \leq 0.05$.

Regarding time, students were significantly faster completing exercise 2 with Jayvee than with Python. These results could indicate that a DSL can make routine data engineering tasks, as often found in open data sources, easier as the exercise mainly requires basic data validation and transformations. However, while not statistically significant, it seems noteworthy from Figure 3 that the median time needed for exercise 5 (handling GTFS files and filtering by id) was higher in Jayvee than in Python. From interviews, we understand that while dealing with ZIP files is easier in Jayvee than Python, filtering data by an ID is not well-supported as of now.

Students' impressions of difficulty and quality of result were answered on 5-point Likert scales and are plotted as diverging stacked bar charts [40, 41]. To calculate the median, we mapped them to numbers from 1 (*Very easy/Very low*) to 5 (*Very hard/Very high*).

Responses to the perceived difficulty of the exercises are plotted in Figure 4, and details are shown in Table 5. For difficulty, the alternative hypothesis is:

H_A^{Diff}: *The difficulty of solving the exercise is significantly lower using Jayvee compared to using Python.*

Exercise 3 is a notable outlier because only a few students who used Python responded. In contrast, more students who used Jayvee answered and reported a high perceived difficulty in Jayvee. In addition, the free text feedback highlighted missing features in Jayvee for the deleting of multiple, not adjacent columns that made the exercise on changing data structure hard. We noted this feedback and included it in our follow-up interviews.

We found that students had significantly less difficulty solving exercise 4 using Jayvee than Python. From later interviews, it became clear that students struggled with the non-standard format of the CSV data for exercise 4, where multiple measurements for one device are concatenated in one row. When using Pandas to load the CSV into a dataframe, a multi-index is automatically created, which is complicated to remove. These problems show that, while often helpful, automation can introduce challenges, and a careful balance between hidden logic and explicit modeling has to be found. The interviews confirmed that this trade-off

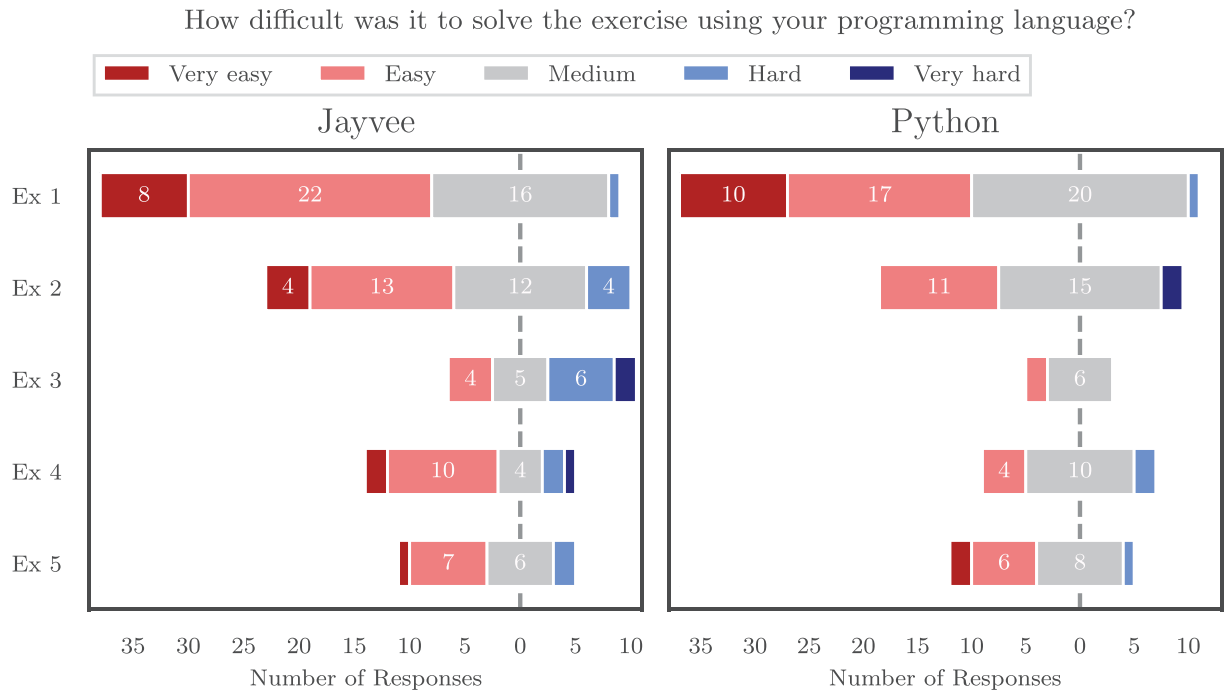


FIGURE 4 | Impressions of the difficulty of completing the exercise, depending on the language used (lower is better).

TABLE 5 | Sample size, median and Mann–Whitney U and p-value for the difficulty of exercises for H_A^{Diff}.

Exercise	n_{jv}, n_{py}	Mdn_{jv}, Mdn_{py}	U	p (less)
Ex1	47, 48	2.0, 2.0	1086.5	0.372
Ex2	33, 28	2.0, 3.0	397.5	0.158
Ex3	17, 8	3.0, 3.0	93.0	0.943
Ex4	19, 16	2.0, 3.0	102.0	**0.040***
Ex5	16, 17	2.5, 3.0	141.0	0.584

*$p \leq 0.05$.

played a major role in the exercise's perceived ease when solved with Jayvee.

Similarly, responses about the quality of the resulting pipeline are shown in Figure 5, and details can be found in Table 6. The alternative hypothesis for quality is:

H_A^{Qual}: *The quality of results is significantly higher when using Jayvee compared to using Python.*

We found statistically significant differences between Jayvee and Python with libraries regarding impressions of the quality of the resulting data pipeline for exercise 1. It is visible from Figure 5 that this difference primarily is caused by some students considering the quality of the Python as low. Here, the large amount of hidden logic that comes from using Pandas might have led to the impression of less control over the data pipeline logic, as discussed in later interviews.

In summary, the data shows that no significant difference exists regarding previous experience with programming, Python, or Jayvee between the two groups that completed the data engineering tasks. Between languages, participants had significantly more experience with Python than Jayvee.

Nevertheless, individual exercises were completed with statistically significant improvements regarding reported speed, difficulty, or quality of result for Jayvee. This indicates that students were able to learn Jayvee to an adequate level quickly and use it successfully to complete data engineering tasks on real open data sets. Significant improvements could be found for challenges that align well with the currently implemented feature set of Jayvee.

These results show that using a DSL like Jayvee is a viable alternative to a GPL with libraries for data engineering tasks (answering RQ 1) as long as the feature set of the DSL is expansive enough. We noticed a spike in perceived difficulty during exercise 3 and planned the follow-up interview study to investigate causal relationships.

5.2 | Interview Study

We conducted exit interviews with volunteers to explore possible explanations for the quantitative survey results and extend them with a description of the effects of using a DSL over a GPL with

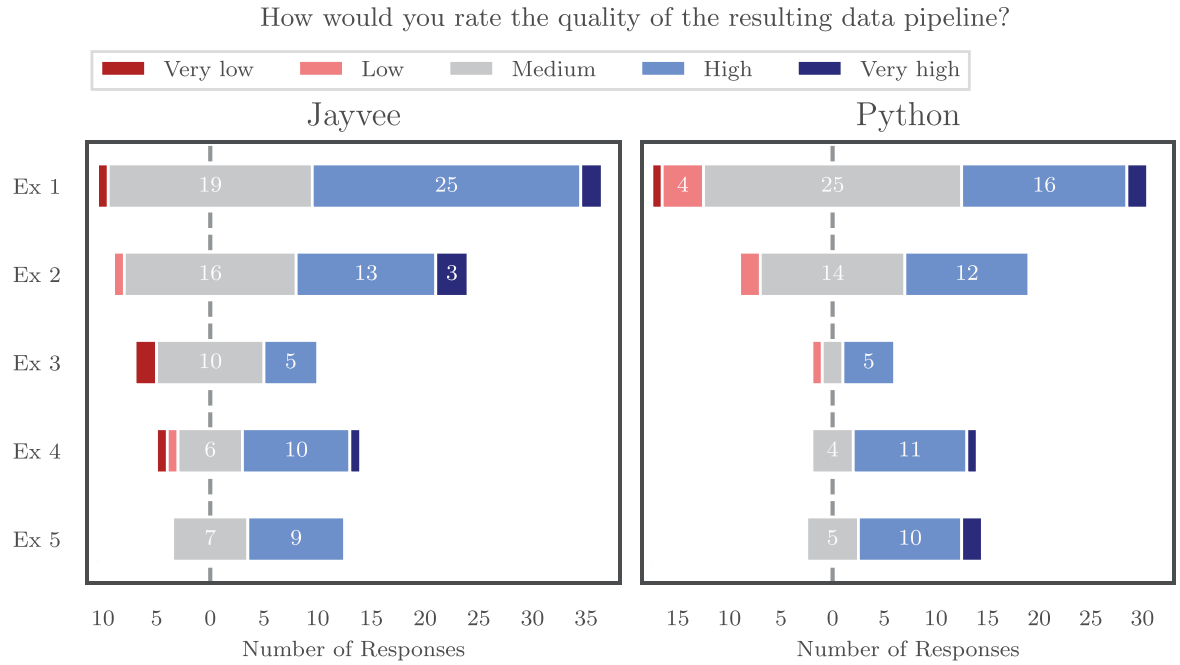


FIGURE 5 | Impressions of the quality of the resulting pipeline, depending on the language used (higher is better).

TABLE 6 | Sample size, median and Mann–Whitney U and p-value for quality of exercise results for H_A^{Qual}.

Exercise	n_{jv}, n_{py}	Mdn_{jv}, Mdn_{py}	U	p (greater)
Ex1	47, 48	4.0, 3.0	1377.0	**0.021***
Ex2	33, 28	3.0, 3.0	515.0	0.200
Ex3	17, 8	3.0, 4.0	47.5	0.911
Ex4	19, 16	4.0, 4.0	122.5	0.873
Ex5	16, 17	4.0, 4.0	107.5	0.884

*$p \leq 0.05$.

libraries to create data pipelines. The transcribed interviews and free-text answers from post-exercise surveys were analyzed using thematic analysis according to Braun and Clarke [34] as described in Section 4. Because the participants' impressions could be influenced by their previous experience, we conducted a course exit survey, asking for self-assessments of their experience in programming, Python, and Jayvee again after completing the data engineering course. The results are shown in Table 7 to provide additional context to participants' quotes.

The resulting themes from the interviews were grouped into three higher-level themes, as summarized with the thematic map in Figure 6:

- *Participants' impressions of speed, difficulty, and quality* of their exercises in Jayvee and Python. This topic most closely relates to Jayvee itself and expands on the results of the quantitative data to answer RQ1 and RQ2.
- *Effects of using a DSL over a GPL with libraries* consists of themes relating to the general effects of using a DSL instead of a GPL on challenges, workflows, and artifacts like source code created by participants, directly related to RQ3.
- *Considerations for a new data engineering language* include themes that do not directly compare using a DSL with a GPL. Instead, it summarizes lessons learned when developing a new language in the domain of data engineering.

TABLE 7 | Experience of interview participants after completing the data engineering course.

Participant	Programming (of 10)	Python (of 5)	Jayvee (of 5)
S0	8	4	5
S2	7	4	3
S3	6	3	2
S5	9	5	4
S7	7	3	3
S8	8	3	3
S10	9	4	4
S11	7	2	4

5.2.1 | Speed

Comparisons of implementation speed between Python and Jayvee were rarely made, with Python mostly being preferred if they did. Participants noted that Jayvee is fast to use for problems that fit its domain well, but can be complicated for more complex data sets or tasks outside its feature set. This aligns with the results of the post-task surveys that showed a statistically significant difference between the time needed to complete a simple data pipeline setup in Jayvee, but indicated that an exercise with challenges outside the feature set of Jayvee was slower to solve.

Execution performance was not considered a problem, even though Jayvee is considerably less optimized than Python. The missing concerns about execution speed were remarkable on their own. However, the data engineering tasks focused on our use case of batch processing smaller datasets, as often found in open data contexts. We interpret this as a sign that execution performance is less relevant to users' perceptions once an acceptable baseline is met in this specific context only. We assume that

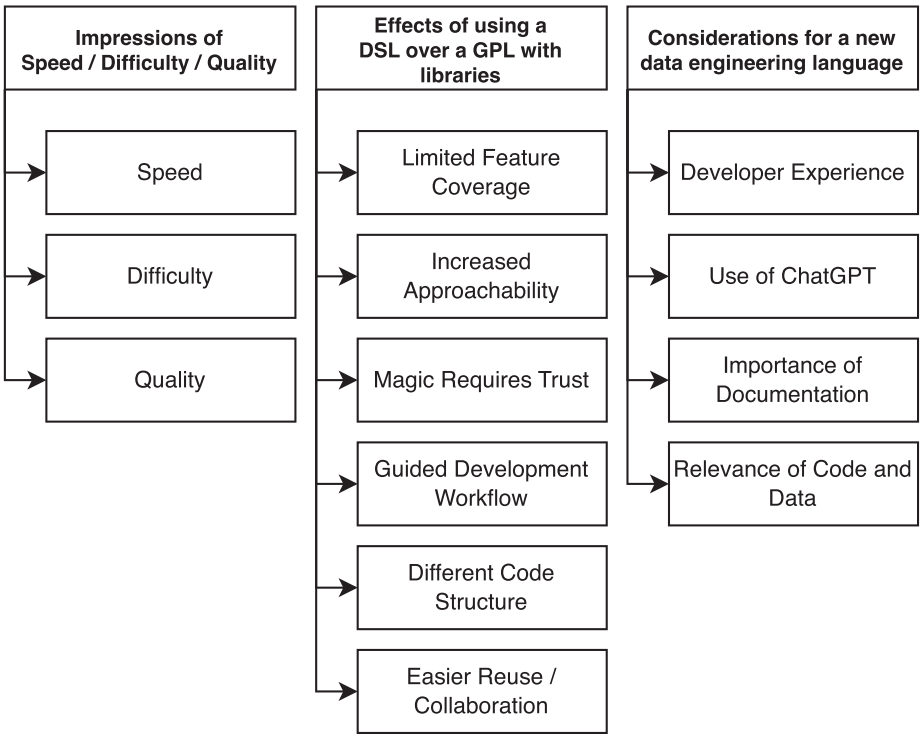


FIGURE 6 | Thematic map from thematic analysis of students' interviews and survey responses.

in domains with larger datasets, the differences in optimization between Jayvee and Python would introduce challenges.

5.2.2 | Difficulty

Generally, students considered Jayvee easy to use and fast and easy to learn. Contributing to this experience was especially the limited scope of the DSL, which means there are fewer options to learn. As S2 puts it: "But for me, it was a bit confusing first in pandas because there are so many options. So I think if you do it the first-time, it's easier in Jayvee." A similar view is expressed by S11: "[. . .] you compare Python, which has a lot of functionality, with Jayvee with limited functionality. The problem is the more you customize, the more complicated it gets." Additionally, students pointed out that previous experience in data engineering made it easy to get started with Jayvee, meaning previous domain knowledge can be leveraged to lower the barrier of entry to get started with implementation.

Specific to the exercises, it became clear from the surveys that exercise 3 was unusually difficult to solve in Jayvee, and we took note to follow up in the interviews. Students explained that the difficulty was due to missing features for working with unconnected columns in a datasheet, this issue is discussed in more detail in Section 5.2.4.

Regarding the lower difficulty of solving exercise 4 in Jayvee, it became clear that loading data into dataframes with Python/Pandas can lead to complications stemming from hidden assumptions (described in more detail in Section 5.2.8). Additionally, working with ZIP files was identified as easier in Jayvee than in Python, showing the potential of a DSL to support a limited number of highly relevant file types in the domain it covers well and to enable their use.

5.2.3 | Quality

As for any software artifact, the quality of data pipelines has multiple dimensions. Overall, students evaluated the quality of data pipelines written in Jayvee positively, but mainly focused on understandability. Students found data pipelines in Jayvee easier to read than Python, especially for non-programmers. S0 points out: "Even if someone who does not know anything about programming languages would read this data pipeline, they would understand [. . .]. They would automatically understand what's going on."

The main reason that was identified was that pipelines written in Jayvee allow readers to get a good overview. The pipes and filters structure enforces creating an explicit hierarchy or sequence of what steps are executed in what order: "What made the quality good is that you have a good overview of what exact task is happening after which, like there is kind of a hierarchy. It starts with the first block, then the second block, and they have specific names and so on, so you have a way [. . .] better overview than Python because everything has a hierarchy.", (S3).

Students also described how this enforced structure provided guidelines and reminders on what to consider while implementing their data pipelines, leading to a higher-quality final result. This is especially notable in light of exercise 1 showing significant improvements in perceived quality because Jayvee enforces the explicit assignment of value types for the extracted data, while Python with Pandas encourages users to rely solely on automated assignments that might change if the underlying data changes.

Further effects of the changed development workflow are also discussed in more detail in Section 5.2.7.

5.2.4 | Limited Feature Coverage

Limited feature coverage can be caused by both missing features that have yet to be implemented and features that might not have a place in a DSL at all. A DSL can be much easier to use for the limited use cases it covers but suffers from being difficult to use outside of them, as S11 mentions: "[. . .] the main advantage of Jayvee is if you have an easy use case, you can write a pipeline down really fast. I think if it gets complex, then you have to look to find your own workaround."

Regarding not yet implemented features, students experienced this issue with exercise 3, which required changing the data structure by deleting multiple, not adjacent columns—while Jayvee only supports deleting single or adjacent columns as of now. Accordingly, we received negative feedback about the missing features, and exercise 3 was perceived as considerably harder than the others (see Figure 4).

Aside from not yet implemented features, students with backgrounds as software developers pointed out that it is unclear how to handle cross-cutting concerns for data pipelines like monitoring or testing in Jayvee. A Python script might send a Slack message for monitoring or logging an intermediate result to Kafka, and it was unclear how to approach these challenges in Jayvee. These requirements do exist for the operation of data pipelines as software artifacts, but they are not part of the domain of data engineering itself. For any DSL, it is a question of whether these cross-cutting concerns should be part of the language design itself and, if so, to what extent. One potential solution to the cross-cutting concerns and extendability of a DSL would be to enable the execution of GPL code, an option that we heavily discussed internally and assumed would feature prominently in the interviews. Surprisingly, this suggestion was only made by one of the interview participants.

5.2.5 | Increased Approachability

The limited feature set of a DSL strongly affects its approachability in the two dimensions of programming experience domain knowledge.

Regarding needed programming experience, participants reported a strong divide between Jayvee's smaller and all-in-one feature set and the mature library ecosystem of Python. Having all functionality as part of the language allows for one central, compact source of information in the form of online documentation, which was generally preferred: "it's better as you have

one central source of information and you don't have that much where you don't find what you need.", (S11).

At the same time, many possible libraries and implementation approaches can exist in a GPL like Python that lead to fragmentation in communities and sources of information that require more experience to navigate. Especially for Python, libraries are complex and have to be learned like a separate language. Students mentioned they knew how to use, for example, Pandas instead of how to program in Python itself and having trouble understanding code from other libraries. Researching fitting libraries was described as time intensive and requires expert knowledge of Python and its libraries, for example by S11: "[. . .] my main criticism about Python, is you have to know which library you use. If you don't, then you have a lot of work to do. [. . .] you can write very good and very compact code and a few lines and get much, but you have to know what you're doing."

In the same quote, the positive side of the effect of programming experience is mentioned: Experienced programmers can leverage their knowledge into using a GPL with libraries well and write short and performant code that solves a problem elegantly. In this sense, a DSL has a lower skill floor, that is, can be used by novice programmers with less previous programming knowledge to solve a problem, but also a lower skill ceiling for professional programmers.

Domain knowledge greatly influences the approachability of a DSL compared to a GPL. Domain experts can reuse their existing knowledge to understand DSL code, and students drew the comparison of Jayvee code to data pipelines multiple times.

Another comparison was made to spreadsheet software like Excel or Google Sheets, for example, by S7: "[. . .] when I use Jayvee, I can think [of] the data pipeline, like I am using Excel. Yes, I am using Excel and then I can think like that and use this to create a pipeline [. . .] but when I use Python, I must think I am a developer or I am a data engineer." When asked why they had this impression, students pointed to the cell selection syntax that is modeled after spreadsheet software (e.g., 'A1-A3' to select the first row and first to the third column, instead of index-based access in Python with Pandas) and to the fact that Jayvee splits working on data shape (using 2D string data structures called 'Sheets') from assigning value types instead of combining both in dataframes.

This similarity to spreadsheet software is relevant because some students reported that their previous experience with data engineering was not from programming but massaging data in, for example, Excel. For other domains with mainly smaller, sheet-based datasets (like many open data domains), this could allow subject-matter experts to translate their existing experience with spreadsheet software into familiarity with Jayvee, similar to the students.

Moreover, working with Jayvee also had a positive effect on related skills, like data pipeline architectures, and the knowledge could also be transferred to Python. S2 explains: "It's now more clear how to structure a data pipeline. [. . .] And I think after programming in Jayvee, I saw in switching to Python, I saw more the structure of the Python code.". Similarly, S5 adapted their Python code after getting exposed to the pipes and filters approach of Jayvee: "I actually explored your block and pipe concept [. . .] I tried to write my project on this concept. So I tried to write this block and pipe in Python as a class.".

5.2.6 | Different Code Structure

The effects of using a DSL over a GPL with libraries on code structure are mainly caused by the strong structure of small, connected, and named blocks of logic that Jayvee enforces. This structure is compared to Python code, written in good style with named functions as described by S11: "In Python, I also tried to modularize my code. [. . .] What you do in Jayvee with pipes is, in general, what I would say is a good method to modularize your code. What I also would expect in another programming language." Because this style is essentially enforced by the DSL, implementation in Jayvee is described as less flexible but more structured than Python.

With inexperienced programmers or scripts that should 'just run,' data pipelines in more flexible languages can be difficult to maintain as S11 goes on: "I think we often see ugly Python code that just runs, but that's not very good maintainable in the end. It's not very abstract written. It just should run." Of course, the tradeoff for enforced structure is that implementation can take longer if all that is needed is a one-off script.

In addition to the difference in structure, students also experienced an effect of how dense Python code can be compared to Jayvee blocks. One line of code, for example, opening a remote CSV file using `read_csv` in Pandas, can lead to the execution of complex logic that has the potential for many different types of errors (in this example, from network issues opening a remote file to parsing errors because of ill-formatted CSV). Because of this density, students described Python code as difficult to debug, as it was unclear where an error occurred and which of the many options to adapt.

In contrast, Jayvee's pipes and filters architecture creates smaller units of code (in blocks) that belong together. This positively affected debugging, making it easier to locate the source of an error. In addition, by enforcing the colocation of related code, it was easier to understand the whole context of a section of a data pipeline. S3 describes the difference to Python: "It's grouped [in Jayvee]. In Python, you could write in the first line, have your dataset variable, and then in line 15 finally work with it to delete rows and so on [. . .]".

5.2.7 | Guided Development Workflow

The different development workflow of students when creating data pipelines followed from their approach to improving a dataset: They worked from the source data by narrowing (e.g., by removing columns and rows or restricting value types) and did not consider working backward from a goal state they wanted to achieve. In fact, Jayvee includes a block that selects columns from a dataset based on an allowlist approach that was described as confusing because students did not understand how to delete columns with it.

Descriptions of the implementation process in Python were uniform: Students optionally started by outlining their approach with comments and wrote imperative code to achieve their goal first, then refactored their script as needed. The implementation process in Jayvee was described less uniformly, though most students defined blocks first and connected them to a pipeline in a final step.

However, students highlighted that the structure that the pipes and filters architecture enforces helped them by providing a guideline of what to do and an order to do it in. S10 describes the process as: "But here [in Jayvee] you have to extract data, then you have to call the interpreted file [. . .]. There were protocols you have to follow first, then you can transform the data.".

In addition to providing guidelines for the structure of the data pipelines, students also experienced the individual blocks as reminders of which steps needed to be implemented in their pipeline, as summarized by S0: "the very streamlined approach of Jayvee that leads you through the steps basically [. . .] it allows you to always think of, maybe I should do some validation here. Maybe I should put some constraints on the data.". These reminders changed the development workflow because they forced developers to think about, for example, assigning value types explicitly to columns of data that might be automatically assigned by type inference in libraries like Pandas.

5.2.8 | Magic Requires Trust

Hidden logic that was described as "magic" in Python/Pandas versus the explicit definitions in Jayvee introduced a tradeoff between magic and trust towards the data pipelines and their results. S2 expresses the feeling as: "If it [Jayvee] compiled, I got most of the things I programmed. If it compiled I got the data I wanted [. . .] but compared to Python there were no big issues if it compiled. I think it was less like magic. In Python you use a function, it's magic in the background. And in Jayvee, it was more like, I know what happened.". The tradeoff described by the participants was that more automated functionality (or 'magic') also means less trust in the correctness of the output data.

The reasons for this effect are that magic can (and does) go wrong but does not produce an error during the execution of the pipeline but only results in an unexpected result. In addition to the time needed to implement a data pipeline, participants regularly needed to verify that the output was what they expected until they were satisfied. An additional effect is that it is difficult to fix if the 'magic' goes wrong. This can occur, for example, with unusually formatted CSV data that leads to Pandas creating a multi-index when creating a dataframe. When this happens, it is much harder to work around the automation than to just not use any automation at all, especially with a library as complicated as Pandas.

The downside of more explicit definitions was identified as more verbose code and slower implementation speed. With the pipes and filters architecture, if the individual steps are too small, they will reduce how fast a pipeline can be created. A potential solution would be compositions of often used functionality, as suggested by S8: "Sometimes for the stuff you would expect people to do very often, an aggregate would have been easier."

5.2.9 | Easier Reuse / Collaboration

Lastly, participants described how using a DSL affected the reuse of and collaboration on pipeline code, with the pipes and filters architecture identified as supporting collaboration and reuse of code. S8 describes this as: "You could reuse the existing pipelines really well because you had most often needed the same steps for input and output. So if I want to ingest some stuff, I can reuse some blocks [. . .]". Of course, reusing code in blocks is similar to extracting parts of an imperative data pipeline into functions and reusing those in Python. However, the flexibility of Python as a GPL with many libraries was described as a challenge to reuse and collaborate because collaborators might use different implementations or libraries that do not work with each other. Additionally, knowledge barriers exist if other developers use different libraries from the ones the participant has experience with.

The use of user-defined value types instead of if-statements for data validation had an additional, positive effect on reuse, as S10 explains: "If you want to follow the constraint in Python, we have to introduce if statements, but here [in Jayvee] you have to create your own data type and you can reuse it. So that was a plus point for Jayvee [. . .]". Using appropriate value types, defined by subject-matter experts, to document and validate data can be a strength of a domain-specific language. An important consideration is the ease of use to create, use, and share these value types so that they are preferred over filtering values with if statements.

5.2.10 | Developer Experience

Regarding *considerations for a new data engineering language*, participants commented on the developer experience of Jayvee as a new language. While a few participants pointed out that the IDE support could be improved by better autocompletion, overall feedback regarding the provided extension for VSCode was sparse or, in some cases, even positive. Providing good IDE support out of the box by implementing a new language using a tool like Langium proved to be a strength of Jayvee. However, students pointed out that they would have liked file templates and scaffolding for what is considered a good code style in Jayvee to improve the IDE experience further.

Challenges with tooling experienced by participants include version confusion between documentation, interpreter, and VSCode extension, as well as difficulty debugging Jayvee code. With a fast-changing new language, it is of high importance to establish clear error messages for version mismatches or an automated way to update to new versions early. During the exercises, we made one new release of Jayvee that introduced confusion, as S2 describes: "it showed the error on Visual Studio, but it worked if I ran it on the command line". Other participants had similar issues with mismatched versions between the different tools.

A large challenge experienced when implementing data pipelines in Jayvee was difficulty debugging. Participants asked for clearer error messages and requested a debugging tool. While Jayvee

does provide basic console debugging outputs using a command line flag, initially students did not find out about this optional parameter. We recommend enabling debug output by default (and providing an opt-out if not needed) and carefully considering their error messages. Regarding error messages, an additional concern is that the smaller community of a new language makes searching for explanations of error messages more complicated.

5.2.11 | Importance of Documentation

Fewer community resources raise the importance of documentation. We provided documentation in the form of a website that documents core concepts and details such as block descriptions. While the documentation was generally appreciated by students, we learned that including it with the IDE support would have improved the experience. S0 points out: "one minus point compared to Python is that Python has all this documentation integrated into the IDEs. So I just hover over some library and I get some information on it in the IDE and don't have to navigate somewhere.". Their sentiment was generally shared by participants, who reported not liking to read the documentation itself and preferring smaller, targeted documentation to their use case directly in the IDE.

In addition to the way the documentation is provided, content and structure are the most important qualities. Regarding structure, related content should be interlinked instead of just presenting a list of language concepts (like blocks). For content, aside from the basic syntax definition, good documentation mainly needs examples and has to ensure those examples are complete. S2 describes their problems with incomplete examples in Python documentation "[. . .] then the example stopped at some point and it took a lot of time for me to get from the point that the example stopped to my own implementation [. . .]", pointing out that having to work with incomplete examples can be slow and frustrating. Other content requests included tutorials for common use cases and more documentation for error messages.

5.2.12 | Use of Chatgpt

The increased use of AI tools like ChatGPT to assist with programming shows how new technology can introduce new language requirements. Some students reported using ChatGPT for research ("How can I develop this? Then ChatGPT will tell you.", S7), to generate starting solutions ("So ChatGPT also recommends some solution.", S5) or even as a debugging tool ("[. . .] we are not getting that clear errors from that. And I tried to search [. . .] on ChatGPT as a tool", S10).

Because Jayvee is a new language, ChatGPT does not provide any usable answers for questions about it—in large contrast to mature languages like Python, which are well-supported. While the use of ChatGPT might not be an important consideration in a classroom or academic context, developers of new languages should consider how they can support development with code generation or LLM-based AI tools in the future.

5.2.13 | Relevance of Code and Data

Lastly, a topic of consideration unique to data engineering is the relevance of data in addition to source code while implementing a data pipeline. In the context of data pipelines, code is only relevant in combination with the data it manipulates. Students struggled to work with Jayvee because it did not support looking at the intermediate data between each step of a data pipeline. Suggested solutions include a `print` statement or supporting the use of Jayvee in Notebooks, a common environment to develop data pipelines in Python.

In this regard, the automated type inference for columns in Pandas dataframes was also pointed out as helpful because it provides hints about the underlying data. New languages in the data engineering domain should consider this requirement and make it as easy as possible to visualize the data flowing through a data pipeline while implementing it, ideally with data summaries or automated type inference instead of showing the raw data only.

6 | Discussion

To the best of our knowledge, the empirical insights presented in this work are among the first to explore how working with a DSL based on pipes and filter concepts effects data engineering. We therefore captured the diversity of effects of using a DSL, instead of deeply exploring one specific aspect or feature. As a result, we consider the insights presented here important, but as a start of a succession of multiple studies that investigate individual effects in more detail.

The chosen population of master students related to computer science, AI, and data science is a good proxy for members of open data communities who have some previous exposure to programming but are not professional software engineers. As a result, we assume that the results obtained will generalize well to the work of practitioners in open data contexts who do not have a background in software engineering. Still, an important research opportunity exists in gathering more empirical data about how open data practitioners work with data and especially how their success is affected by different tools.

However, more in-depth work is needed to learn what effect individual DSL features, like the pipes and filter architecture chosen by Jayvee, have on the work with the DSL and if they are the best choice. To strengthen the generalizability of the findings, more narrow comparisons of individual implementation decisions with comparable features in GPLs are needed.

Nonetheless, the results indicate that it is possible to quickly learn a new DSL for data engineering and use it to build data pipelines with little previous experience. The main reason for this effect is the reduced complexity and scope of a focused DSL in comparison to a GPL with libraries. Tasks outside the DSL's feature set can become challenging or even impossible to solve. For this reason, it will be important to carefully plan the scope of the DSL to cover its domain without introducing too much complexity again.

In the open data context, batch processing small files with tabular data covers a large part of existing data sources [29, 30]. However, to be able to improve all data sources, further types of data and

modes of operation will need to be introduced. Improving human performance in building high-quality data pipelines is one important part of building tools. For smaller data sets, execution performance is less relevant. With a DSL as structured as Jayvee, implementers have to write more readable code which leads to higher-quality results, especially for novice programmers. Expert software engineers might still want to work with a GPL with libraries to be more flexible, but a DSL can enable subject-matter experts to contribute as well.

A theme that emerged from interviews was the general challenges when introducing a new language for data engineering, such as the need for a good debugger or the importance of documentation. The feedback shows that designing and implementing a new language is not only an academic challenge, but must be supported by a surrounding ecosystem if there should be any hope of serious adoption.

This mirrors the experience from other external DSLs, such as model transformation languages, as discussed in Section 2. The quality of the ecosystem and tooling that surrounds a language is essential to its use by practitioners, with editors, debuggers, and validation or analysis tools described as essentials [17]. In our data, we also find problems with missing debuggers. However, editors are rarely mentioned as an issue and sometimes even highlighted positively. The reason is probably that most participants used the VSCode plugin we provided, the development of which was fairly straightforward because it relies on the autogenerated support for the language server protocol (LSP) provided by Langium.

Nonetheless, it remains an open question whether implementing an external DSL is the best approach to take. By building an internal DSL based on a popular host language, such as Python, the existing tooling of the host language could be reused. Modern GPLs have improved considerably compared to older versions and lower the productivity gap between DSLs and GPLs even for domain-specific tasks, as investigated by Höppner et al. [42] for Java in the domain of model transformations. In their study, the domain-specific requirement of tracing was the major influence on whether a DSL reduced complexity. For data engineering, it would be important to investigate if similar processes exist that can introduce a large overhead to implement with GPLs but could be automatically handled by a DSL.

New ways of programming, such as using AI support from tools like ChatGPT, are rapidly changing the way novice programmers work. The way LLMs and other AI tools can interact with a language should be actively planned. Structured DSLs might have an advantage over GPLs in this regard because they are more limited and therefore easier to reason about.

Finally, with collaborative data engineering already being a growing practice [7, 43], reducing entry barriers for participants who are not software engineers can be a stepping stone toward a higher amount of collaboration in open data engineering.

7 | Limitations

As a mixed-methods study, multiple viewpoints are relevant to set the results into context. We discuss the limitations and mitigations we took for the quantitative data gathered in descriptive surveys according to the well-known framework of threats to validity as discussed in Wohlin et al. [39]. For the qualitative results from the interview study, we use the trustworthiness criteria described by Guba of credibility, transferability, dependability, and confirmability [44].

However, while we present potential limitations from both viewpoints, employing data and method triangulation by using a mixed-method research design strengthens the results by allowing one method to reduce the weaknesses of the other.

7.1 | Threats to Validity

We evaluate potential threads to validity regarding the quantitative results of the descriptive surveys according to the classification presented in Wohlin et al. [39].

Threats to conclusion validity are challenges to drawing the correct conclusions about relationships between the treatment and results. The measures we have taken for our analysis reflect the subjective experience of the participants and are not objective, automated measurements and must be interpreted in that context. To reflect this, we have taken care to label references to the measures as participants' impressions instead of objective truths. Combined with the additional context provided by the interview study, we consider these insights still appropriate for the exploratory nature of this study; however, more rigorous follow-up studies in more controlled settings are needed to confirm our measurements.

An additional threat lies in the potential heterogeneity of the participants as students, especially since they come from different master's degrees. However, the variance in degrees provides a more realistic setting and allows us to discuss the effect of using a DSL with insights from various backgrounds. To reduce the impact of this threat, we've compared the previous experience of participants with a course entry survey (shown in Figure 2) and found no statistically significant difference between groups.

Because the authors of this study are also involved in creating the DSL that was investigated as treatment, bias and searching for positive results is a clear threat to conclusion validity. To mitigate this, we defined the complete research design as well as hypotheses ahead of data collection and used standard research designs and statistical tests. We committed to and reported the full, partially negative results of all hypothesis tests. Nonetheless, subconscious bias remains a threat to conclusion validity. Therefore, we have published an accompanying data release and invite replication by independent researchers.

Internal validity describes the extent to which a design can mitigate outside influences on the outcome that are unknown to the researcher, such as bias, apart from the treatment. Because participants did solve the exercises in their own programming environments, outside influences aside from the programming language

are a concern. We chose this approach because of the exploratory nature of the research and to increase the generalizability of the results by allowing participants to use the tools they would for a real task. As a consequence, additional research in a more strict setting, like controlled experiments, would be needed to increase the rigor of the results.

The selection of volunteers out of a class of students might influence the results because volunteers are generally more motivated to solve new tasks than the general population. Additionally, students might be biased toward responses in favor of Jayvee if they suspected a positive influence on their grades. We mitigated this threat by using anonymized surveys while clearly communicating to the students that we would not analyze the data before grades were published.

Construct validity is concerned with how well the research construct represents the underlying concept or theory under evaluation. Mono-method bias might be a concern for the descriptive survey results because only one measurement was taken for each construct. In the larger context of the complete study, this concern is mitigated by the additional context provided by qualitative feedback from the mixed-method design.

A social threat to construct validity presents itself in the fact that it was reasonably easy to guess the hypotheses under test because participants were aware of the questions regarding speed, difficulty, and quality after answering the first survey and knew that other students were using a different language to solve the exercises. However, with the anonymous and optional surveys, there was no pressure on participants to conform or skew their answers to either side.

External validity describes the ability to generalize the results of a design to different settings, such as from data among students to industry. Using students as participants is a threat to external validity and limits how much the insights can be generalized. This threat is mitigated by the fact that the student population for this study was from the masters level and, therefore, more experienced. When using students as an approximation, it is important to clearly understand which population is being represented [25]. We chose students because they are non-professional programmers with limited experience in creating data pipelines, so they are similar to subject-matter experts working with data in industry. We consider them good proxies for this population, and we expect the results to generalize well to this limited population. In contrast, we caution against generalizing the results to other contexts, such as professional software engineers or subject-matter experts without any previous exposure to programming.

7.2 | Trustworthiness Criteria

We discuss the qualitative results from the interview study according to the trustworthiness criteria described by Guba [44] of credibility, transferability, dependability, and confirmability.

Our sampling strategy presents a limitation to credibility, as both the online surveys after exercises and the interviews were opt-in and voluntary, potentially leading to a bias of participants who enjoyed working with Jayvee or faced comparatively few challenges. To counteract this, two of the authors spent whole semesters in prolonged engagement while teaching the students and directly experienced their questions about the exercises. We mitigated the risk of attracting students who wanted to please us to improve their grades by making the exercise surveys anonymous and executing the interviews after the students received their grades. Furthermore, we used data and method triangulation by gathering quantitative data about student experiences with the data engineering tasks first and then following up with qualitative data from interviews to confirm and extend the findings. We limited the influence of our presence and behavior on the interviewees by sticking to a predefined interview guide and keeping a neutral tone. We applied thematic analysis as a systematic data analysis approach and mitigated potential confirmation bias by conducting a peer debriefing session with researchers from another university.

Transferability, the degree to which the results can be transferred to other contexts, is limited by only evaluating the use of one specific DSL. We acknowledge that more studies are needed to confirm more general insights. Thus, we limited statements about results to Jayvee or placed appropriate disclaimers when we made inferences about DSLs in general.

The use of students as participants means the study is not directly transferable to professional contexts. However, we consider the suitability of students as a proxy for open data practitioners to be high because most open data practitioners are also non-professional programmers with some previous experience in data engineering.

The exercises were deliberately chosen to represent basic data engineering tasks in both languages: data extraction from the web as single files or via compressed zip archives; data cleaning by removing invalid values and filtering columns; transforming values; and loading the data into a sink. Generalizations beyond this scope for more complex data engineering tasks like combining data of different sources cannot be drawn without further research. We also provided thick descriptions of the identified themes, coupled with direct quotes from the interviews, so that future transfers to other contexts are supported.

Regarding dependability, our goal was to report as much of the research process and data as possible, so the research context is clear. We have to limit access to some data, like the original interviews, due to confidentiality agreements. We have made the complete data and code that were used in the writing of this paper available during the review process, and cited extensively from the interviews to support our conclusions in the qualitative data analysis. In addition, we sought to increase dependability with regular external feedback for individual parts of the research and the presentation of all intermediate results and the research design to wider audiences (see Table 2).

We established the confirmability of the findings through regular peer debriefings and discussions among the authors, in addition to data- and method triangulation. We mitigated the risk of selective observations during the interviews by using an internally reviewed interview guide. Nevertheless, because the authors have also implemented Jayvee, we have to acknowledge our own bias. We consulted external feedback in a peer debriefing session with

reviewers from another university to reduce the risk of a biased viewpoint for analysis. We welcome independent future work to verify our findings.

8 | Conclusions and Future Work

To summarize, we introduced and empirically evaluated a domain-specific language for the creation of data pipelines by comparing it with a general-purpose programming language with libraries. To answer if the use of a DSL has potential during data engineering, we asked *Is using a DSL for data engineering a viable alternative to a GPL with a data engineering library?* and *What is the user's perception of difficulty and quality of results using a DSL compared to a GPL with libraries?* During the period of two semesters, two cohorts of students built typical data pipelines for real-world tabular open data. Data gathered in after-task surveys shows that, even though participants have statistically significantly less experience in Jayvee than in Python, Jayvee provides some significant differences in greater speed, lower difficulty, and higher quality of the resulting pipeline in specific exercises. We therefore conclude that a DSL with a pipes and filter structure can be a viable alternative to a GPL with libraries for data engineering for novice programmers.

Extending the quantitative results, we describe causal relationships to answer *What are the effects of using a DSL for data engineering compared to a GPL with libraries?*, by extracting common topics from participants' interview transcripts using thematic analysis.

We find the more strictly enforced source code structure of a DSL to be a major effect. On the one hand, perceived pipeline quality is higher when implemented with a DSL, especially for novice programmers who might otherwise struggle to structure their GPL code appropriately. Additionally, a consistent structure acts as a helpful guideline during implementation. Specific design choices, such as using blocks and validating data using user-defined value types instead of if statements, enable code reuse and better collaboration. On the other hand, reduced flexibility means pipelines can take longer to implement because one-off script-style implementations are no longer possible. Functionality outside the feature scope of the DSL or cross-cutting concerns such as monitoring are difficult to implement without a GPL. Using a DSL for data engineering is therefore advisable when implementing a data pipeline that is supposed to be of high quality and operated long-term. For one-off data cleaning tasks or requirements outside the scope of the DSL, a GPL should be preferred.

In this context, the decision between an internal and external DSL has to be made. While building an external DSL provides the largest amount of control and potential to perfectly capture the domain, it is also a large programming effort. Tool availability (such as editors and debuggers) as well as tool quality are challenges to external DSLs that impact internal DSLs less because they can reuse existing tooling of the host language. Writing good documentation is required for both internal and external DSLs, and should include complete examples and be available inside the editor. Developers of DSLs must keep this in mind and plan their workload accordingly. However, in our experience, modern language development tools such as Langium make it possible to provide good editor support using the language server protocol with relatively low overhead and make implementing external DSLs possible even with small teams. Due to the nature of the LSP as an open protocol, the first plugins for Jayvee for other editors, such as neovim are already being developed. We conclude from this that tooling is an essential area that should be considered and planned for when implementing a new DSL. Whenever possible, developers should rely on open protocols and provide their own plugins for popular IDEs.

The recent rise in AI tools to support development has important effects on the use of DSLs for data engineering as well. Especially non-professional programmers use new technology like ChatGPT to support them during development, and future languages must take these changed requirements into account. Other important workflows in the domain of data engineering that language developers should keep in mind include Notebook programming, which enables users to tinker with a pipeline while being able to see source code and data at the same time.

When implemented with a DSL, the output of data pipelines is trusted as being correct more than for GPLs. Potential reasons for this are the more consistent structure, together with less hidden logic (less 'magic') and more explicit definitions. Combined, these lead to an increased understanding of what happens during pipeline execution; however, detailed insight would require additional data.

An opportunity for DSLs in data engineering is their ability to allow users to use knowledge outside of software development. This makes the DSL more approachable, especially for non-professional programmers, by requiring less previous experience to evaluate libraries or learn language concepts. In the case of data engineering, previous experience with sheet software is both very common and relevant. By using domain concepts like cell selection syntax that follows the syntax of sheet software like Excel, entry barriers for non-professional programmers could be reduced. While we have found evidence for this effect in the interviews, further work is required to find out the extent of this effect.

Additional research, such as more empirical studies with open data practitioners and other non-professional programmers, is required to better generalize the findings presented in this study. While we consider students a close proxy for hobbyist participants in data engineering for open data, it is important to verify this assumption and extend the insights to professionals in open data contexts.

In future work, we plan to evaluate individual features of a DSL for data engineering in detail in controlled experiments. More rigorous, quantitative evaluations of individual features will strengthen the insights from this initial, qualitative validation of the effect of DSLs in data engineering. By investigating individual features, future implementation of DSLs can support data engineering efforts more effectively.

Author Contributions

The authors take full responsibility for this article.

Acknowledgments

This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17045 Software Campus 2.0 (Friedrich-Alexander-Universität Erlangen-Nürnberg) as part of the Software Campus project 'JValue-OCDE-Case1.' Responsibility for the content of this publication lies with the authors.

The authors thank the anonymous reviewers for their comprehensive feedback that improved the quality of the manuscript. Open Access funding enabled and organized by Projekt DEAL.

Disclosure

The authors have nothing to report.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are openly available in Zenodo at 10.5281/zenodo.14730736. The original interview transcripts are not available publicly due to privacy restrictions, but have been made available to peer reviewers. They are available on request from the corresponding author.

Endnotes

[1] https://github.com/jvalue/jayvee.

[2] https://langium.org/.

[3] https://jvalue.github.io/jayvee/docs/category/block-types.

[4] https://jvalue.github.io/jayvee/.

[5] https://mobilithek.info/.

References

1. I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino, *Data Wrangling: The Challenging Journey From the Wild to the Lake* (Asilomar. people.cs.uchicago.edu: In, 2015).

2. A. Zuiderwijk, M. Janssen, and C. Davis, "Innovation With Open Data: Essential Elements of Open Data Ecosystems," *Information Polity* 19, no. 1,2 (2014): 17–33, https://doi.org/10.3233/IP-140329.

3. P. Heltweg and D. Riehle, "A Systematic Analysis of Problems in Open Collaborative Data Engineering," ✻ACM✻*Transactions on Social Computing* 6, no. 3-4 (2023): 1–30, https://doi.org/10.1145/3629040.

4. M. Fowler and R. Parsons, *Domain-Specific Languages* (Addison-Wesley Educational, 2010).

5. U. Leser, M. Hilbrich, C. Draxl, et al., "The Collaborative Research Center FONDA," *Datenbank-Spektrum: Zeitschrift fur Datenbanktechnologie: Organ der Fachgruppe Datenbanken der Gesellschaft fur Informatik e.V* 21, no. 3 (2021): 255–260, https://doi.org/10.1007/s13222-021-00397-5.

6. M. R. Crusoe, S. Abeln, A. Iosup, et al., "Methods Included: Standardizing Computational Reuse and Portability With the Common Workflow Language," *Communications of the ACM* 65, no. 6 (2022): 54–63, https://doi.org/10.1145/3486897.

7. J. Choi, Y. Tausczik, *Characteristics of Collaboration in the Emerging Practice of Open Data Analysis* (ACM, 2017).

8. C. Liu, A. Usta, J. Zhao, S. Salihoglu, "Governor: Turning Open Government Data Portals Into Interactive Databases," in *No. Article 415 in CHI'23* (Association for Computing Machinery, 2023): 1–16.

9. A. Bogatu, N. W. Paton, M. Douthwaite, and A. Freitas, *Voyager: Data Discovery and Integration for Onboarding in Data Science* (OpenProceedings.org, 2022).

10. M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys (CSUR)* 37, no. 4 (2005): 316–344.

11. T. Kosar, S. Bohra, and M. Mernik, "Domain-Specific Languages: A Systematic Mapping Study," *Information and Software Technology* 71 (2016): 77–91.

12. L. M. Do Nascimento, D. L. Viana, P. A. Neto, D. A. Martins, V. C. Garcia, and S. R. Meira, "A Systematic Mapping Study on Domain-Specific Languages," in *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)* (Xpert Publishing Services, 2012), 179–187.

13. D. S. Kolovos, R. F. Paige, T. Kelly, and F. A. Polack, "Requirements for Domain-Specific Languages," in *Proc. of the 1st ECOOP Workshop on Domain-Specific Programming Development (DSPD)* (2006).

14. S. Meliá, C. Cachero, J. M. Hermida, and E. Aparicio, "Comparison of a Textual Versus a Graphical Notation for the Maintainability of MDE Domain Models: An Empirical Pilot Study," *Software Quality Journal* 24 (2016): 709–735.

15. T. Kosar, N. Oliveira, M. Mernik, et al., "Comparing General-Purpose and Domain-Specific Languages: An Empirical Study," *Computer Science and Information Systems* 7, no. 2 (2010): 247–264.

16. A. N. Johanson and W. Hasselbring, "Effectiveness and Efficiency of a Domain-Specific Language for High-Performance Marine Ecosystem Simulation: A Controlled Experiment," *Empirical Software Engineering* 22 (2017): 2206–2236.

17. S. Höppner, Y. Haas, M. Tichy, and K. Juhnke, "Advantages and Disadvantages of (Dedicated) Model Transformation Languages: A Qualitative Interview Study," *Empirical Software Engineering* 27, no. 6 (2022): 1–71, https://doi.org/10.1007/s10664-022-10194-7.

18. B. Hoffmann, N. Urquhart, K. Chalmers, and M. Guckert, "An Empirical Evaluation of a Novel Domain-Specific Language - Modelling Vehicle Routing Problems With Athos," *Empirical Software Engineering* 27, no. 7 (2022): 180, https://doi.org/10.1007/s10664-022-10210-w.

19. K. Klanten, S. Hanenberg, S. Gries, and V. Gruhn, "Readability of Domain-Specific Languages: A Controlled Experiment Comparing (Declarative) Inference Rules With (Imperative) Java Source Code in Programming Language Design," in *Proceedings of the 19th International Conference on Software Technologies* (SCITEPRESS–Science and Technology Publications, 2024).

20. T. Kosar, M. Mernik, and J. C. Carver, "Program Comprehension of Domain-Specific and General-Purpose Languages: Comparison Using a Family of Experiments," *Empirical Software Engineering* 17 (2012): 276–304.

21. T. Kosar, S. Gaberc, J. C. Carver, and M. Mernik, "Program Comprehension of Domain-Specific and General-Purpose Languages: Replication of a Family of Experiments Using Integrated Development Environments," *Empirical Software Engineering* 23, no. 5 (2018): 2734–2763, https://doi.org/10.1007/s10664-017-9593-2.

22. D. Garlan and M. Shaw, "An Introduction to Software Architecture," in *Advanceson Software Engineering and Knowledge Engineering*, vol. 2 (World Scientific, 1993), 1–39.

23. M. Shaw and D. Garlan, "Formulations and Formalisms in Software Architecture," in *Computer Science Today: Recent Trends and Developments*, ed. V. J. Leeuwen (Springer Berlin Heidelberg, 1995), 307–323.

24. R. B. Johnson, A. J. Onwuegbuzie, and L. A. Turner, "Toward a Definition of Mixed Methods Research," *Journal of Mixed Methods Research* 1, no. 2 (2007): 112–133, https://doi.org/10.1177/1558689806298224.

25. D. Falessi, N. Juristo, C. Wohlin, et al., "Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments," *Empirical Software Engineering* 23, no. 1 (2018): 452–489, https://doi.org/10.1007/s10664-017-9523-3.

26. V. A. Thurmond, "The Point of Triangulation," *Journal of Nursing Scholarship: An Official Publication of Sigma Theta Tau International Honor Society of Nursing / Sigma Theta Tau* 33, no. 3 (2001): 253–258, https://doi.org/10.1111/j.1547-5069.2001.00253.x.

27. S. Spall, "Peer Debriefing in Qualitative Research: Emerging Operational Models," *Qualitative Inquiry: QI* 4, no. 2 (1998): 280–292, https://doi.org/10.1177/107780049800400208.

28. J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "*Measuring Programming Experience*," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)* (IEEE, 2012), 73–82.

29. J. Umbrich, S. Neumaier, and A. Polleres, "*Quality Assessment and Evolution of Open Data Portals*," in *2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud)* (IEEE, 2015), 404–411.

30. J. Mitlohner, S. Neumaier, J. Umbrich, and A. Polleres, "*Characteristics of Open Data CSV Files*," in *2016 2nd International Conference on Open and Big Data (OBD)* (IEEE, 2016).

31. B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software EngineeringLondon*, eds. F. Shull, J. Singer, and D. I. K. Sjøberg (Springer, 2008), 63–92.

32. H. Jansen, "The Logic of Qualitative Survey Research and Its Position in the Field of Social Research Methods," *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* 11, no. 2, Art. 11 (2010), https://doi.org/10.17169/fqs-11.2.1450.

33. H. Kallio, A. M. Pietilä, M. Johnson, and M. Kangasniemi, "Systematic Methodological Review: Developing a Framework for a Qualitative Semi-Structured Interview Guide," *Journal of Advanced Nursing* 72, no. 12 (2016): 2954–2965, https://doi.org/10.1111/jan.13031.

34. V. Braun and V. Clarke, *Thematic Analysis* (American Psychological Association, 2012), 57–71.

35. B. Kitchenham, L. Madeyski, D. Budgen, et al., "Robust Statistical Methods for Empirical Software Engineering," *Empirical Software Engineering* 22, no. 2 (2017): 579–630, https://doi.org/10.1007/s10664-016-9437-5.

36. R. Vallat, "Pingouin: Statistics in Python," *Journal of Open Source Software* 3, no. 31 (2018): 1026, https://doi.org/10.21105/joss.01026.

37. S. S. Shapiro and M. B. Wilk, "An Analysis of Variance Test for Normality (Complete Samples)," *Biometrika* 52, no. 3/4 (1965): 591–611, https://doi.org/10.2307/2333709.

38. H. B. Mann and D. R. Whitney, "On a Test of Whether One of Two Random Variables Is Stochastically Larger Than the Other," *Annals of Mathematical Statistics* 18, no. 1 (1947): 50–60.

39. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering* (Springer Science + Business Media, 2012).

40. N. B. Robbins and R. M. Heiberger, "Plotting Likert and Other Rating Scales," in *Proceedings of the 2011 Joint Statistical Meeting*, vol. 1 (American Statistical Association, 2011).

41. R. Heiberger and N. Robbins, "Design of Diverging Stacked bar Charts for Likert Scales and Other Applications," *Journal of Statistical Software* 57 (2014): 1–32, https://doi.org/10.18637/jss.v057.i05.

42. S. Höppner, T. Kehrer, and M. Tichy, "Contrasting Dedicated Model Transformation Languages Versus General Purpose Languages: A Historical Perspective on ATL Versus Java Based on Complexity and Size," *Software and Systems Modeling* 21, no. 2 (2022): 805–837, https://doi.org/10.1007/s10270-021-00937-3.

43. A. X. Zhang, M. Muller, and D. Wang, "How Do Data Science Workers Collaborate? Roles, Workflows, and Tools," *Proceedings of the ACM on Human-Computer Interaction* 4, no. CSCW1 (2020): 1–23, https://doi.org/10.1145/3392826.

44. E. G. Guba, "Criteria for Assessing the Trustworthiness of Naturalistic Inquiries," *ECTJ* 29, no. 2 (1981): 75, https://doi.org/10.1007/BF02766777.

D

Paper 4: Can a Domain-Specific Language Improve Program Structure Comprehension of Data Pipelines? A Mixed-Methods Study.

In this appendix, the paper *Can a domain-specific language improve program structure comprehension of data pipelines? A mixed-methods study.* is reproduced in full. The manuscript is currently submitted and undergoing peer review. A preprint is published as:

Heltweg, P., Schwarz, G.-D., & Riehle, D. (2025). Can a domain-specific language improve program structure comprehension of data pipelines? A mixed-methods study. In arXiv [cs.PL]. arXiv. http://arxiv.org/abs/2505.16764 [19]

[1] *https://creativecommons.org/licenses/by/4.0/*

CAN A DOMAIN-SPECIFIC LANGUAGE IMPROVE PROGRAM STRUCTURE COMPREHENSION OF DATA PIPELINES? A MIXED-METHODS STUDY.

PREPRINT

Philip Heltweg
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
philip@heltweg.org

Georg-Daniel Schwarz
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
georg.schwarz@fau.de

Dirk Riehle
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
dirk@riehle.org

June 3, 2025

ABSTRACT

In many application domains, domain-specific languages can allow domain experts to contribute to collaborative projects more correctly and efficiently. To do so, they must be able to understand program structure from reading existing source code. With high-quality data becoming an increasingly important resource, the creation of data pipelines is an important application domain for domain-specific languages.

We execute a mixed-method study consisting of a controlled experiment and a follow-up descriptive survey among the participants to understand the effects of a domain-specific language on bottom-up program understanding and generate hypotheses for future research.

During the experiment, participants need the same time to solve program structure comprehension tasks, but are significantly more correct when using the domain-specific language. In the descriptive survey, participants describe reasons related to the programming language itself, such as a better pipeline overview, more enforced code structure, and a closer alignment to the mental model of a data pipeline. In addition, human factors such as less required programming experience and the ability to reuse experience from other data engineering tools are discussed.

Based on these results, domain-specific languages are a promising tool for creating data pipelines that can increase correct understanding of program structure and lower barriers to entry for domain experts. Open questions exist to make more informed implementation decisions for domain-specific languages for data pipelines in the future.

Keywords program comprehension · data pipelines · data engineering · domain-specific languages · mixed-methods study · open data

1 Introduction

Domain-specific languages (DSLs) can be a useful alternative to general-purpose programming languages (GPLs) in many application domains. By focussing on one domain, they can have a reduced scope and re-use glossary and concepts from the application domain, making them easier to learn and more efficient to program for domain experts (Kosar et al., 2018; Johanson and Hasselbring, 2017). However, because DSLs are a specialized tool, they have to be carefully evaluated to determine whether they provide enough benefits to make their adoption a good choice.

When working on non-trivial software applications, developers must first understand the program structure from source code. Only then can they make changes to extend existing implementations or fix bugs. Program comprehension, in general, is estimated to be the dominant activity while programming, with more than 50% of time spent (Roberto Minelli and Lanza, 2015; Xia et al., 2018). Therefore, the effects of a DSL on program structure comprehension are essential for the usefulness of a DSL in an application domain.

The evaluation of DSLs generally has to be domain-specific (Kosar et al., 2018). Increasingly, high-quality data, and with it data engineering, is of large importance in industry because many innovative apps and AI applications rely on access to data. Sources for data sets vary from company internal data to open data, with open data mainly published by governments but also by some private entities.

Depending on the type of data, creating an automated data pipeline is a major part of data engineering. An example is regularly changing data, such as schedules released as open transport data, that should be ingested and improved automatically with updated releases.

In complex domains, data-engineers must collaborate with subject-matter experts to understand the meaning of data. A common challenge during these collaborations is that subject-matter experts lack programming experience, which complicates it to find a shared collaboration artifact with professional programmers (Heltweg and Riehle, 2023).

Domain-specific languages can be a useful middle-ground, that enables subject-matter experts to contribute directly to the creation of data pipelines, as previously shown in other domains (Johanson and Hasselbring, 2017; Lopes et al., 2021).

DSLs can be grounded in the formal and informal glossary of domain experts, such as sketches (Wile, 2004). A common mental model for a data pipeline is a graph of processing steps connected by pipes, known from visual programming. A DSL can provide an explicit syntax and semantics to express this data pipeline structure with the pipes and filters architecture.

In previous explorative work, we found using a domain-specific language based on this architecture had positive effects on speed, quality of the solution and perceived difficulty when solving data engineering exercises on real life open data sets (Heltweg et al., 2025).

Building on this high-level validation, we aim to understand how domain-specific languages contribute to improved performance by subject-matter experts and what language features are important in more detail. To do so, we conduct a series of empirical evaluations using quantitative and qualitative methods. Previous research shows that programming language research lacks empirical studies, instead focusing on solution proposals (do Nascimento et al., 2012). However, empirical user studies to evaluate usability are essential tools that can lead to insights that would not have been gained otherwise (Buse et al., 2011; Barišić et al., 2018).

When contributing to a collaborative data engineering project, the first thing a subject-matter expert will need to do is read and understand the intention behind data pipeline source code. To start, we therefore focus on bottom-up program comprehension, the process of inferring the intentions behind an implementation from reading source code (Wyrich et al., 2023); we do so in the domain of building data pipelines by non-professional programmers (subject-matter experts).

In the context of this mixed-methods study, we compare data pipelines implemented in a DSL using an explicit pipes and filters architecture (Jayvee) to imperative scripts in a GPL with libraries for data engineering (Python with Pandas). We performed an initial controlled experiment to gather quantitative data on task performance in terms of time and correctness. In a follow-up survey, we look for causal influences for the experiment outcomes..

With the results, we answer the following research questions:

Research Question 1: Do data pipelines implemented in Jayvee change bottom-up program structure comprehension compared to Python/Pandas for non-professional programmers...

a: regarding speed?

b: regarding correctness?

c: regarding the perceived difficulty?

Research Question 2: What reasons exist for effects on program comprehension for data pipelines implemented in Jayvee compared to Python/Pandas for non-professional programmers?

In this article, we contribute:

1. A mixed-methods approach, combining a controlled experiment with a descriptive survey, to evaluate the effects of DSLs in the domain of data pipeline modelling.
2. Quantitative data, based on a controlled experiment, on how strongly the use of a DSL in the domain of data pipeline modelling can influence pipeline structure understanding, contributing to the growing literature on domain-specific languages and motivating their use in data engineering.
3. Explanations for these effects from participant surveys to guide future practitioners or researchers that implement DSLs for data engineering.

2 Related Work

Empirical research into the effects of domain-specific languages has been performed across multiple domains. Kosar et al. have used controlled experiments to compare DSLs with GPLs and libraries. Initially, in the context of GUI programming, they compared the DSL XAML with C# Forms, with XAML performing better for answering questions on provided source code (Kosar et al., 2010).

With a similar approach, Kosar et al. (2012) extended the insights to the domains of feature diagrams and graphical descriptions, again comparing a DSL with a GPL and an appropriate library. While the previous experiments were performed on paper, a replication study in Kosar et al. (2018) allowed the use of IDEs. In all studies, participants performed more accurate and efficient in program comprehension tasks using a DSL than a GPL with libraries.

Similar to our work, Kosar et al. have evaluated the use of DSLs for different domains using experiments and note that because DSLs are domain-specific, they must be evaluated for each domain. Our goal is to extend their work with the domain of creating data pipelines for data engineering. In addition to a purely quantitative comparison of performance, we also provide qualitative insights into potential reasons for different performance.

Other DSLs with similar structure, either for data pipelines or using blocks, have been proposed. Cingolani et al. (2015) present an external DSL for the creation of data pipelines in the domain of biological data called BigDataScript. They similarly plan to support subject-matter experts, but do so by replicating script-style programming and abstracting from the underlying architecture. In contrast to our work, they demonstrate the independence towards architecture, robustness, and scalability of the language implementation technically but do not evaluate it empirically.

PACE is an external DSL for continuous integration pipelines with a block structure that compiles to JSON, presented in Fonseca et al. (2020). In a controlled experiment, participants are tasked with pipeline creation and extension while thinking aloud, comparing PACE with their previous system of manually creating JSON configs with the results showing an improvement using PACE. We use a similar mixed-methods research design, however, in a very different context (understanding data pipelines by non-professional developers instead of creation of CI pipelines in an industrial setting).

In their PhD thesis, Misale (2017) designed and developed PiCo, a DSL based on pipes and the data flow computational model. They demonstrate the capability of their design and evaluate the performance of the implementation using case studies and experiments with Flink and Spark. In comparison, our work provides an empirical evaluation of code comprehension instead.

As with our study, students are commonly used as participants in controlled experiments, which can provide useful data if their use as a proxy for a specific type of developer is appropriate (Falessi et al., 2018).

Lopes et al. compared a text-based DSL with a graphical tool in a different domain (entity-relationship modeling) with students (Lopes et al., 2021). Their results are aligned with ours, showing that a textual approach using a DSL is possible with a slight advantage in quality but no difference in effort. A similar controlled experiment on readability (speed and correctness) of type inference rules shown in a DSL or Java implementation is described in Klanten et al. (2024). The authors point out that research into programming language design lacks empirical studies, a research gap our work contributes to reducing.

Hoffmann et al. evaluated Athos, a DSL that targets subject-matter experts in the domain of vehicle routing and traffic simulation, compared to JSpirit (Java with libraries) (Hoffmann et al., 2022). As with the previous studies, the DSL improved efficiency. In addition, participants reported improved user satisfaction when using Athos. Even with the planned end users being subject-matter experts, the authors rely on students as proxies for subject-matter experts.

Similar to these studies, our work uses students as participants because we consider them a good approximation for practitioners that had first programming experiences but are not professional developers (such as subject-matter experts that have to do data engineering).

Empirical evaluations of DSLs with subject-matter experts are rare. An example is Johanson and Hasselbring (2017) in which ecologists use the Sprat Ecosystem DSL and the GPL C++ to solve program comprehension tasks related to ecosystem simulations. Participants are subject-matter experts from a non-technical domain (marine science) with only moderate previous programming experience. Time to task completion and correctness were measured, with the tasks being solved in less time and with higher correctness using the DSL. The context of our research are also subject-matter experts and not technical users. We extend the insights gathered in this study by investigating a different domain (the creation of data pipelines).

3 Methods

We used a mixed method research design (Johnson et al., 2007), combining quantitative data from a controlled experiment according to Ko et al. (2015) and a descriptive survey according to Kitchenham and Pfleeger (2008) with qualitative data from free-text responses to the same survey. We chose thematic analysis according to Braun and Clarke (2012) to extract common themes from the survey responses. An overview of the complete research design is shown in Figure 1.

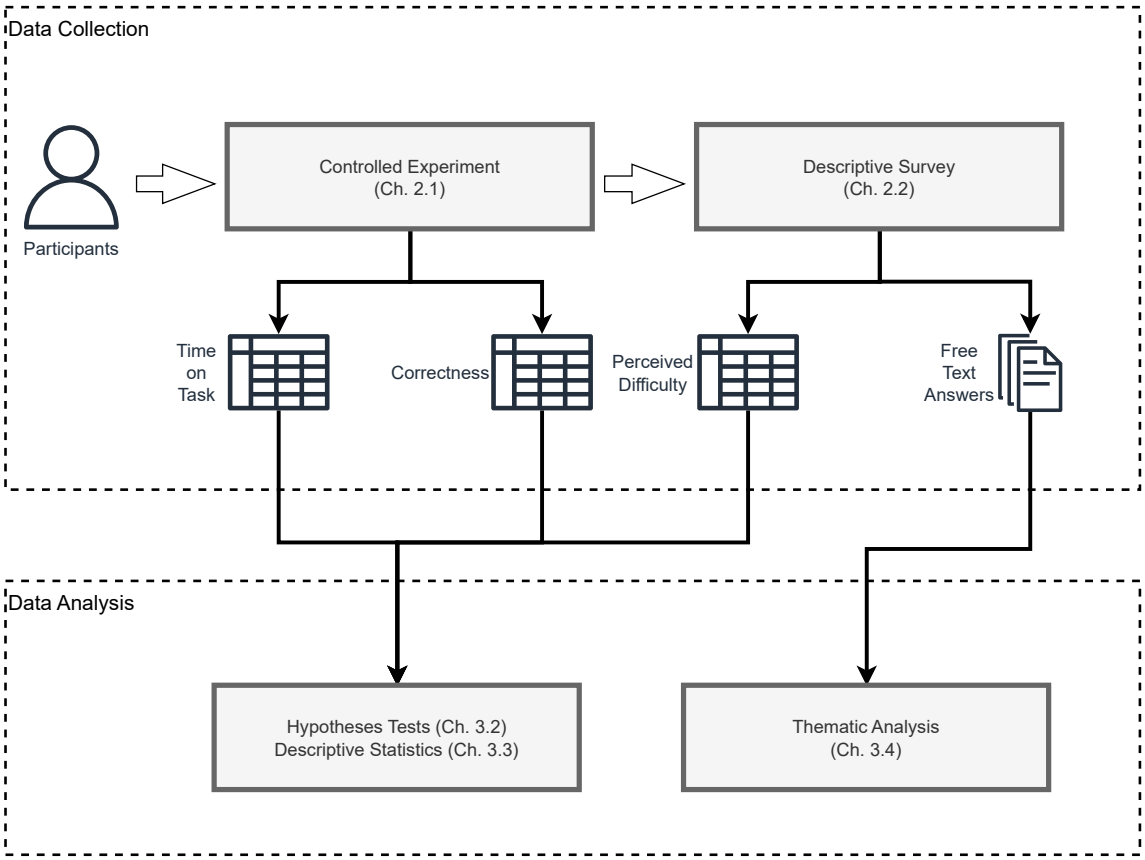


Figure 1: Overview of the mixed method research design, split into data collection and data analysis.

The combination of these methods allows us to validate our hypotheses in a rigorous manner and uncover potential causal relationships that strengthen the insights and enable us to generate further hypotheses to test in future work. Additionally, the qualitative responses also touch other topics of program comprehension in addition to program structure, allowing us to describe a wider diversity of effects regarding *RQ2*.

3.1 Jayvee, a Domain-Specific Language for Data Pipelines

Jayvee is a DSL for data engineering following the well-known pipes and filters architecture described in Garlan and Shaw (1993) and Shaw and Garlan (1995). The language is designed to align as closely as possible with the mental model of data pipelines as directed acyclical graphs of processing steps, thereby making it easier for subject-matter experts to use than traditional GPLs.

The main elements of a Jayvee pipeline model is a top level `pipeline`, consisting of multiple `blocks`, each representing a processing step. The inputs and outputs of these blocks are connected using `pipes`. Blocks have an `oftype` relationship with `blocktypes`, which defines the input and output types of the block as well as its properties that can be configured.

Jayvee is an external DSL that is not embedded in a host programming language but has its own syntax and semantics. The syntax is implemented using a context free grammar language provided by Langium[1] while a TypeScript based interpreter acts as a reference implementation for the language semantics.

Listing 1 shows an example of a data pipeline implemented in Jayvee. The pipeline consists of three `blocks`, each performing a step in the data processing. At the top of the pipeline definition (line 2-4), the pipeline structure is defined by connecting the blocks using the `pipe` syntax `->`.

```
1  pipeline CarDataPipeline {
2    CarDataCSVExtractor
3      -> CarDataInterpreter
4      -> CarDataSQLiteLoader;
5
6    block CarDataCSVExtractor oftype CSVExtractor
7      url: "https://example.org/data.csv";
8      enclosing: '"';
9    }
10   block CarDataInterpreter oftype TableInterpreter {
11     header: true;
12     columns: [
13       "name" oftype text,
14       // ... further assignments
15     ];
16   }
17   block CarDataSQLiteLoader oftype SQLiteLoader {
18     table: "Cars";
19     file: "./cars.db";
20   }
21 }
```

Listing 1: Data pipeline extracting CSV data and writing it to a SQLite database, written in Jayvee.

Jayvee includes more advanced concepts such as user-defined value types and a standard library of prebuilt, domain-specific blocks. The language is open source and available on GitHub[2], additional documentation is hosted at `https://jvalue.github.io/jayvee`.

3.2 Controlled Experiment

We follow the guidelines on reporting experiments described in Wohlin et al. (2012), originally by Jedlitschka and Pfahl (2005). We first provide informal information about research goals and the context of the experiment, and then report details of the experimental design. The experiment execution and resulting data is reported in section 4.

We followed the Goal/Question/Metric template to define the research objective of the controlled experiment (Wohlin et al., 2012; Basili and Rombach, 1988):

1. Analyze *a DSL and a GPL with a specific data engineering library*
2. for the purpose of *their effect on bottom-up program structure comprehension for data pipelines*
3. with respect to *speed and correctness*
4. from the point of view of *researchers*
5. in the context of *a university course with masters level students learning data science (as proxies for non-professional programmers).*

Our goal was to understand the influence of a DSL on professionals of non-programming disciplines that work with data as part of their jobs. Some examples include data scientists or subject-matter experts, e.g., in biology, that analyze data. Representatives from this population have base programming skills from working with data, but are not professional software engineers.

The experiment was conducted with student participants in person, over two days in computer labs provided by the university. During the experiment, participants solved two program structure understanding tasks by reading source code of a pipeline and recreating the data pipeline structure afterward.

We use a concrete example task as an overview before describing the experiment design in detail in the following sections. Figure 2 is a screenshot of a task view in the web-based experiment tool the participants used. On the left-hand side, under *Pipeline Code* the source code of a data pipeline is shown. This data pipeline was implemented

[1] `https://langium.org/`
[2] `https://github.com/jvalue/jayvee`

either in Jayvee or Python/Pandas, depending on the treatment group. On the right-hand side, under *Pipeline Steps*, participants had to recreate the data pipeline structure by dragging steps from the list of *Unused Steps* into the *Steps in Data Pipeline* and bringing them into the correct order. Once they were satisfied with their solution, they could submit it using the *Submit Solution* button and attempt the next task.

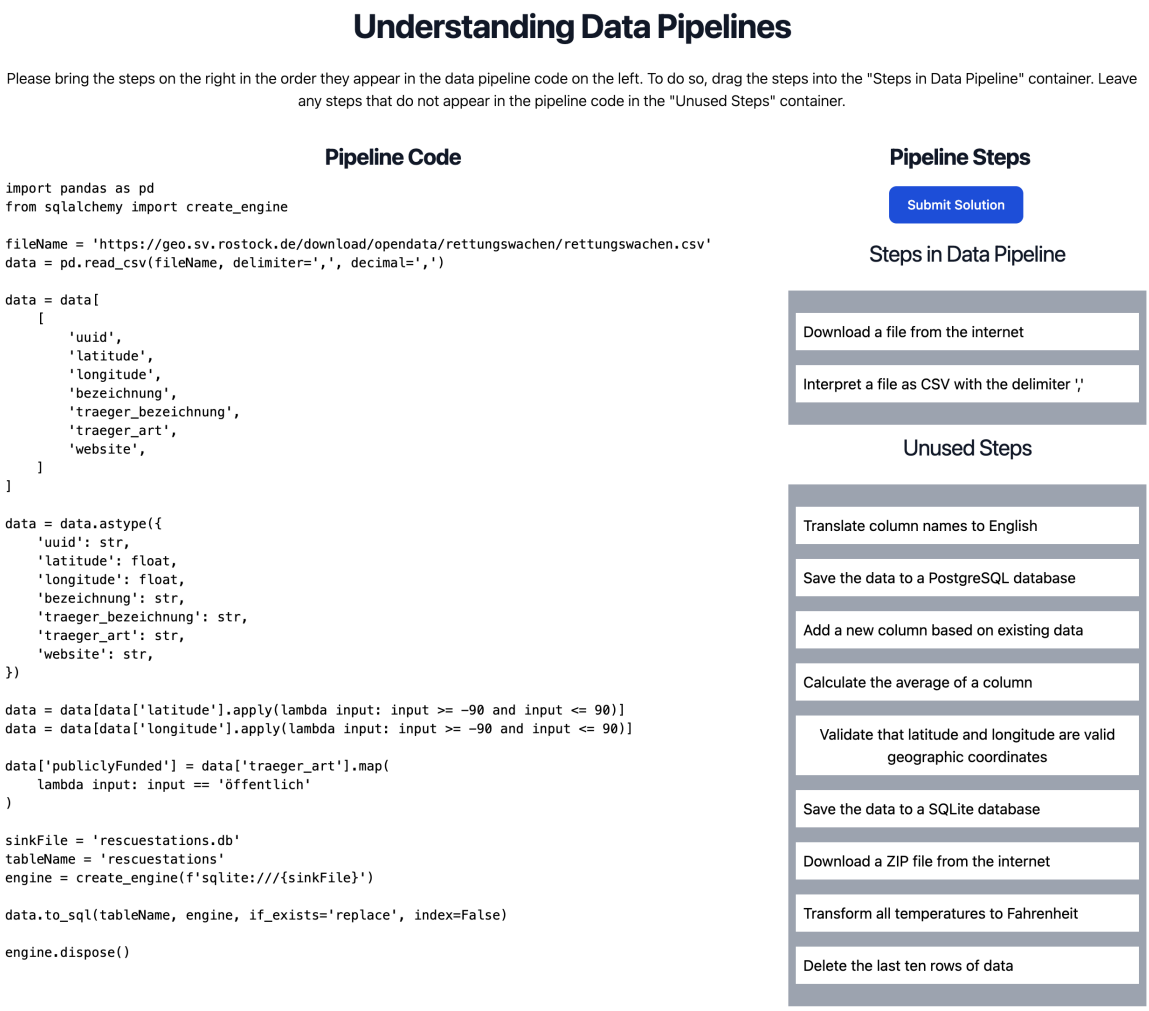


Figure 2: The experiment tool during task 2 in Python/Pandas. Pipeline source code is shown on the left, the recreation using ordered steps on the right.

3.2.1 Goals, hypotheses, and variables

We defined one independent variable, the programming language PL used to implement a data pipeline, either Jayvee (JV) or Python/Pandas (PY).

From the research objectives, we chose time to task completion and correctness as dependent variables. The combination of time and correctness is the most common for comprehension tasks (Wyrich et al., 2023).

Time to completion describes the time between seeing the source code and submitting a solution. At the start of each task, the source code of the data pipeline was hidden so participants could read the available steps they had to categorize and order. We started the time measurement once participants revealed the source code by pressing a button. The time is directly measured in milliseconds by the experiment software and defined as follows:

$$time(PL) : \text{time of submission for task in PL} - \text{time of source code reveal} \tag{1}$$

Correctness is an indirect variable that is calculated from the submitted solution by the participant.

For each task, n potential steps are available for participants to choose from. A subset of these available steps is present in the pipeline, in a specific order. Using the drag and drop interface, participants can categorize steps into *Steps in Data Pipeline* or used steps and *Unused Steps* and decide on an order of steps inside these categories.

To define the correctness of a solution S, we consider two dimensions: Has the participant correctly understood which steps exist in the pipeline source code and have they understood the order in which they are executed?

Regarding *existence*, we count the number of steps that have been categorized correctly, either steps that exist in the pipeline and have been categorized as used or steps that do not exist in the pipeline and have been categorized as unused. Because each step can only be assigned to one category, the maximum number of correctly categorized steps is equal to n.

Regarding *order*, we count the number of swaps needed to bring the steps that the participants categorized as used into the correct order, ignoring incorrectly categorized steps. As a sorting algorithm, we chose selection sort because it requires the minimal number of swaps to sort. We chose to handle the order of steps by sorting instead of comparing with a reference solution because a small error in ordering could mean all following steps are also in the wrong position, which would lead to a large penalty for small errors. In contrast, if the correct order can be reestablished with few swaps, the penalty is more appropriate.

In combination, we can define correctness as follows:

$$correctness(PL) : \frac{\text{\#correctly categorized steps} - \text{\#swaps needed for correct order}}{n} \tag{2}$$

Based on this definition, correctness is a numeric value $[0, 1]$. For example, if a task has 10 available steps, and the participant categorized 9 of them correctly and in the right order, the correctness would be 0.9. If a participant instead categorized all steps correctly, but two swaps were needed to bring the used ones into the correct order, the correctness would be 0.8.

Hypotheses were defined based on the goal to describe effects on speed and correctness.

For *speed*, we defined $H_{0,1}$ as *"Non-professional programmers need the same time to understand the structure of a data pipeline model when implemented in Jayvee compared to Python/Pandas."* with the alternative hypothesis $H_{1,1}$, *"Non-professional programmers do not need the same time to understand the structure of a data pipeline model when implemented in Jayvee compared to Python/Pandas."*. More formally:

$$\begin{aligned} H_{0,1} &: time(JV) = time(PY) \\ H_{1,1} &: time(JV) \neq time(PY) \end{aligned} \tag{3}$$

Regarding *correctness*, we defined $H_{0,2}$ as *"Non-professional programmers understand the structure of a data pipeline model equally correct when implemented in Jayvee compared to Python/Pandas."* with the alternative hypothesis $H_{1,2}$, *"Non-professional programmers can understand the structure of a data pipeline model not equally correct when implemented in Jayvee compared to Python/Pandas."*. More formally:

$$\begin{aligned} H_{0,2} &: correctness(JV) = correctness(PY) \\ H_{1,2} &: correctness(JV) \neq correctness(PY) \end{aligned} \tag{4}$$

3.2.2 Experiment Design

We chose a factorial crossover design according to Vegas et al. (2016) which is a within-subjects design in which each participant is assigned to every treatment exactly once. Crossover designs are well understood and commonly used for software engineering experiments (Wyrich et al., 2023).

The participants completed two tasks reading a data pipeline, implemented in either Jayvee or Python/Pandas and recreating it using a drag and drop interface. We defined two periods (solving task 1 and task 2) and two sequences

AB and BA, see Table 1. Participants were randomly assigned to either sequence without experimenter input, based on a call to JavaScript `Math.random` when they opened the experiment tool. One experiment session included both periods.

Table 1: Factorial crossover design of the controlled experiment according to Vegas et al. (2016)

	Period	
Sequence	Task 1	Task 2
AB	Jayvee	Python/Pandas
BA	Python/Pandas	Jayvee

3.2.3 Participants

The experiment was executed during a masters level course on data engineering and working with open data, offered to students largely studying data science and artificial intelligence as well as some students from computer science and information systems. Because the participants are students and the vast majority of them study degree programs that mainly work with data in a theoretical fashion rather than teach software engineering, they have limited experience programming but have worked on data engineering before. We considered this population an appropriate proxy, as discussed in Falessi et al. (2018), for data practitioners that have some experience with programming but are not professional software engineers.

During the course, students were introduced to Jayvee in two lectures and were encouraged to use Python with Pandas for an individual data science project. The course requires the completion of five data engineering exercises in Jayvee and Python/Pandas, with students switching languages after each exercise. In all lectures that referenced programming challenges, we used examples in Jayvee and Python/Pandas. While we mentioned alternative libraries, we always used Python in combination with Pandas during the module.

We employed convenience sampling from this population by offered students to voluntarily participate in the experiment in place of completing the third homework exercise. Doing so would count as passing the exercise, and enter them into a raffle to win two gift cards of EUR 20 each. If they chose to complete the exercise as normal, they experienced no negative effects, e.g., their grade was unaffected.

3.2.4 Objects, Instrumentation, and Data Collection Procedure

Participants were asked to complete two bottom-up code comprehension tasks in which they had to read the provided source code of a data pipeline and recreate the structure using a drag and drop interface. They completed one task reading a pipeline implemented in Jayvee and one with a pipeline implemented in Python/Pandas, depending on their sequence assignment. Both tasks used a web-based experiment tool (see Figure 2 for a task screen example) and followed the same sequence:

1. Participants were shown the available steps, categorized as unused, while the pipeline source code was hidden.
2. After reading the available steps, participants reveal the pipeline source code using a button press (time measurement starts).
3. Participants drag and drop steps into the *Steps in Data Pipeline* category and bring them in the correct order as they understand the pipeline.
4. When they are satisfied with their solution, participants click on "Submit Solution" (time measurement stops).
5. They are taken to a pause screen where they can start the next task whenever they feel ready.

In addition to time measurements, the experiment tool automatically saved the submitted solution so that correctness could be calculated in the analysis phase. After both tasks, the participants were asked to complete a follow-up survey. The exact version of the tool used by participants can be found online [3].

Both languages were shown as text without syntax highlighting. Two researchers were in the room for every experiment run to monitor the screens of participants and ensure silence. This made sure that participants did not interact with each other or search for solutions on the internet.

For the tasks, we implemented equivalent data pipelines in Jayvee 0.1.0 and Python 3.11 with Pandas 2.0, based on real open data sources.

1. Task 1 is a pipeline that downloads a ZIP-file, extracts it and selects a file as CSV. It then translates some columns names to English, selects a subset of columns and saves the data to a SQLite-database.

2. Task 2 is a pipeline that downloads a file, interprets it as CSV and validates that data in one column are geographic coordinates between -90 and 90. It adds a new column with boolean data, based on another column. Finally, it saves the data to SQLite.

We aligned the code structure as much as possible by implementing each step similarly in Jayvee and script-style Python/Pandas. As an example, Figure 3 compares the source code to extract a CSV file for task 1 in both languages. The example shows the more verbose syntax of Jayvee, utilizing blocks to model processing steps, compared to Python/Pandas. The appendix (section 7) includes a further comparison of source code used in task 2 (Figure 7).

```
HttpDataSource
  ->TextInterpreter
  ->CSVFileInterpreter
  //... further blocks

block HttpDataSource oftype HttpExtractor {
  url: 'https://geo.sv.rostock.de/download/opendata/
       rettungswachen/rettungswachen.csv';
}

block TextInterpreter oftype TextFileInterpreter {}

block CSVFileInterpreter oftype CSVInterpreter {
    delimiter: ',';
    enclosing: '"';
}
```

```
import pandas as pd

fileName = 'https://geo.sv.rostock.de/
    download/opendata/rettungswachen/
    rettungswachen.csv'

data = pd.read_csv(fileName, delimiter=',
    ', decimal=',')
```

Figure 3: Comparison of source code excerpts to extract data from a CSV source, shown for task 1 in Jayvee and Python/Pandas.

We conducted two pilot tests to ensure the data pipeline implementations and the accompanying step descriptions are appropriate and clear. First, we shared the tasks with other researchers that were neither involved in Jayvee development nor the experiment itself. Later, we invited students from previous semesters to take the full experiment remotely while we watched their screen and asked for their feedback afterward. Based on the feedback of both pilot groups, we made minor code and wording adjustments and gained the expectation that the tasks could reasonably be completed in 10 minutes each.

We defined an experiment procedure so multiple experimenters could guide the participants through the following process:

1. Read and acknowledge informed consent information.
2. Open allowed documentation in tabs.
3. Provide an overview about the experiment process, how tasks work and what the experiment measures. Communicate that we expect the experiment to last for roughly 30 minutes and will announce times at 10 minutes and 20 minutes.
4. Solve an initial example task with pseudocode together with participants to familiarize them with the tool.
5. Answer any final questions before asking the participants to start their tasks and no longer interacting with them.
6. Participants complete both tasks and the follow-up survey.
7. Finally, thank the participants and ask them not to share the experiment setup with other participants.

Because we asked participants to submit their own solutions, variations can occur between participants that choose to be faster or more correct, depending on their confidence (Ko et al., 2015). To reduce this effect, we asked the participants to favor correctness over speed if in doubt.

The full source code of both tasks, the experiment procedure and the informed consent handout can be found in the replication package [3].

[3] All links can be found in the Data Availability Statement (section 7).

3.3 Descriptive Survey

We designed a cross-sectional, descriptive survey according to Kitchenham and Pfleeger (2008) to assess how participants perceived the difficulty of understanding the data pipeline from Jayvee code compared to Python/Pandas.

As part of the survey, participants completed an online questionnaire after completing the experiment, with two agreement questions *How difficult was it to understand the data pipeline written in Jayvee?* and *How difficult was it to understand the data pipeline written in Python?*. Answers could be given on a 5-point Likert scale. We assigned numbers from 1 (*Very easy*) to 5 (*Very hard*) to be able to calculate medians and defined $difficulty(PL)$ as the median of the answers for JV and PY respectively.

To answer RQ 1c: Do data pipelines implemented in Jayvee change bottom-up program structure comprehension compared to Python/Pandas for non-professional programmers *regarding perceived difficulty*, we defined $H_{0,3}$ as *"Non-professional programmers do not perceive a data pipeline model as easier or harder to understand when implemented in Jayvee compared to Python/Pandas."* with the alternative hypothesis $H_{1,3}$, *"Non-professional programmers do perceive a data pipeline model as easier or harder to understand when implemented in Jayvee compared to Python/-Pandas."*. More formally:

$$\begin{aligned} H_{0,3} &: \textit{difficulty}(JV) = \textit{difficulty}(PY) \\ H_{1,3} &: \textit{difficulty}(JV) \neq \textit{difficulty}(PY) \end{aligned} \tag{5}$$

In addition, participants were provided free-text input fields for the questions *What makes data pipelines written in Jayvee difficult/easy to understand?*, *What makes data pipelines written in Python difficult/easy to understand?*, and *What are the differences between Jayvee and Python that influence how easy / hard it is to understand data pipelines?*.

To analyze this qualitative data, we chose thematic analysis according to Braun and Clarke (2012). Because we had no preconceived theory but wanted to understand causal relationships for the experiment results, we chose an inductive approach, letting the themes emerge from the data.

During the thematic analysis, we first familiarized ourselves with the data by reading all survey responses in detail.

Afterward, we created codes from the data and constructed a codebook by grouping related codes into themes. Our goal was the creation of a codebook that is clear and themes that can be consistently understood by multiple readers. We therefore worked in iterations, with multiple authors applying the codebook to responses independently and discussing any differences in coding that emerged from unclear descriptions to improve the clarity of themes.

For each iteration:

1. We selected a subset of the responses at random
2. The first author coded the subset of responses and afterward updated the codebook with new insights
3. The updated codebook was shared with another author, who used the codebook to code the same subset of responses
4. The authors met to qualitatively discuss any differences in coding and the clarity of the codebook and the codebook was updated according to the discussion
5. The first author used the updated codebook to re-code all previous responses

Because our goal was to explore the diversity of reasons for the effects on program comprehension, we chose theoretical saturation as a guideline to judge the maturity of our codebook, meaning no or few new insights are gained from analyzing additional data (Bowen, 2008). We counted codes that were assigned to each survey response, as well as any codebook changes (newly created, deleted, moved or updated codes and themes). We consider theoretical saturation to be reached when codebook changes are rare (indicating that the codebook is stable), but codes are still assigned to new responses (indicating that the codebook is relevant to the topic of the response).

4 Results

4.1 Participant Sample

Our sample consisted of 57 volunteers from a masters level course about advanced methods of data engineering that was completed by 98 students. Students mainly came from master's degree programs in artificial intelligence, data science and computer science. At the start of the semester, we used an online survey with previously validated questions by Feigenspan et al. (2012) to measure previous experience in programming generally and Python and Jayvee specifically. Median programming experience was 7 (of 10), median comparison to classmates 3, median experience in Python 4 and median experience in Jayvee 1 (all of 5). At the end of the semester, we repeated the survey and the median experience of course participants in Jayvee had increased to 3 ($n = 77$). A detailed overview of the course entry survey results can be found in Figure 8 (section 7).

After the course entry survey, all participants heard two lectures on Jayvee programming and solved one data engineering exercise in Jayvee as part of the training for the experiment.

Of these 57 participants, 29 were randomly assigned to sequence AB and 28 to sequence BA.

4.2 Hypotheses tests

We used Python 3.11 with Pingouin 0.5.5 (Vallat, 2018) for the statistical analysis of the data. We consider tests at the standard $\alpha = .05$ to be statistically significant.

For each participant, we calculated time on task and correctness as described in subsubsection 3.2.1.

Initially, we performed a Shapiro-Wilk test (Shapiro and Wilk, 1965) to check if the variables were distributed normally. At $\alpha = .05$, both variables were non-normal. As a result, we chose the Wilcoxon signed-rank test (Wilcoxon, 1945) as non-parametric alternative to a paired t-test because it is appropriate for paired data from the crossover experiment (Wohlin et al., 2012; Vegas et al., 2016).

Variable distributions are plotted as kernel-density-plots to give an overview and make it easy to see non-normality (Kitchenham et al., 2017).

We report effect sizes based on the matched pairs rank-biserial correlation (RBC) as an appropriate measure of effect size for the Wilcoxon signed-rank test used for the experiment data (Kerby, 2014). As a correlation, it is equal to the difference between proportions of favorable and unfavorable evidence, with 0 meaning no effect and positive values indicating support for H_1. In addition to RBC, we also report CLES as a more intuitive measure of effect size, first introduced by McGraw and Wong (1992), but based on the generalization by Vargha and Delaney (2000) to allow non-normal and ordinal data such as the survey responses on a Likert scale. We interpret CLES based on the guidelines in Vargha and Delaney (2000) as either small ($\geq .56$), medium ($\geq .64$) or large ($\geq .71$).

4.2.1 Hypothesis 1: Speed

The null hypothesis we defined for speed was $H_{0,1}$: *"Non-professional programmers need the same time to understand the structure of a data pipeline model when implemented in Jayvee compared to Python/Pandas."* We therefore chose a two-sided Wilcoxon signed-rank test, with the results shown in Table 2.

Table 2: Wilcoxon signed-rank test for $H_{0,1} : time(JV) = time(PY)$

n	Mdn_{JV}	Mdn_{PY}	W-val	alternative	p-val	RBC	CLES
57	252.37	234.23	750	two-sided	.546	.093	.52

We have no reason to reject the null hypothesis and accept $H_{0,1}$: *"Non-professional programmers need the same time to understand the structure of a data pipeline model when implemented in Jayvee compared to Python/Pandas."* Based on the data and the underlying distribution (see Figure 9 in section 7), it is reasonable to conclude that the use of programming language had no significant effect on time to completion in either direction.

4.2.2 Hypothesis 2: Correctness

The distribution of correctness for Jayvee and Python/Pandas is plotted in Figure 4.

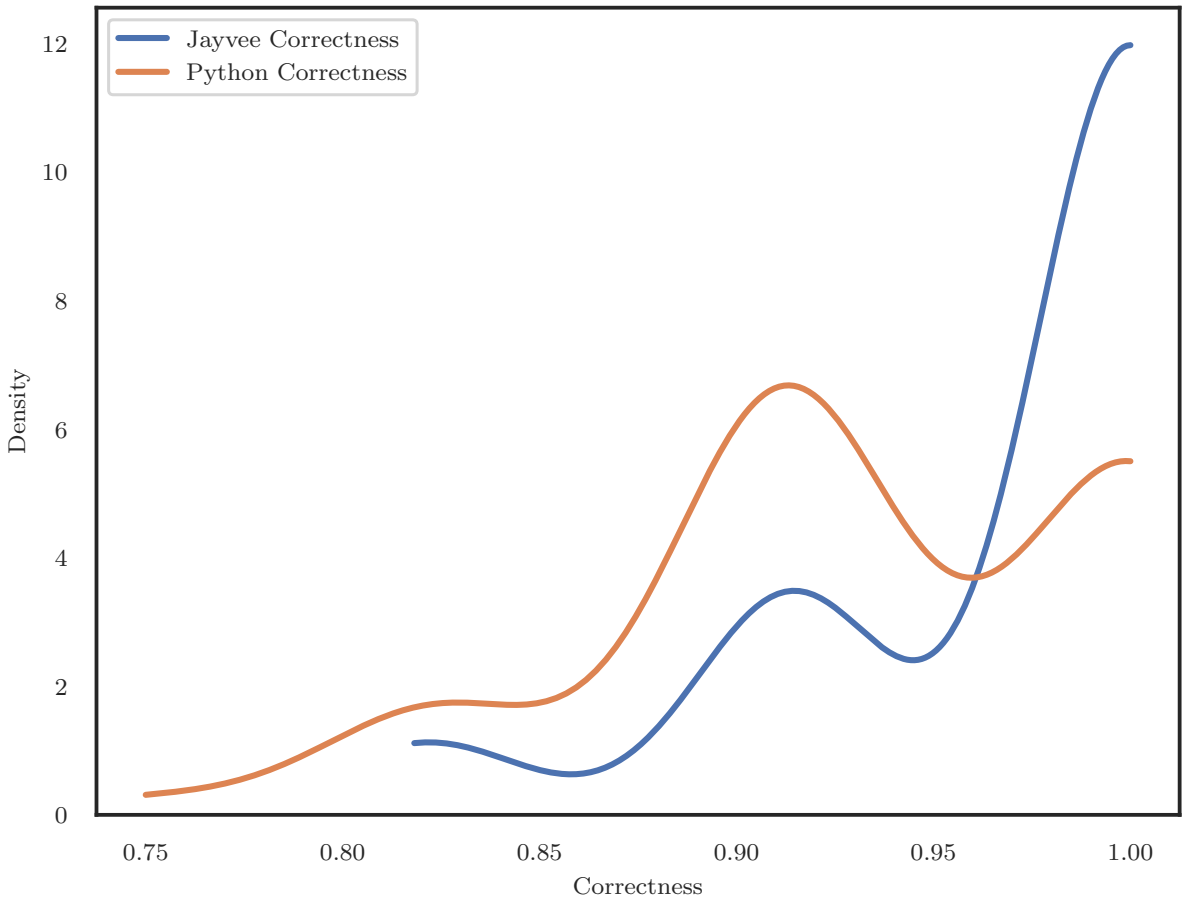


Figure 4: Kernel-density-plot of correctness of solution for Jayvee compared to Python/Pandas.

The null hypothesis we defined for speed was $H_{0,2}$: *"Non-professional programmers understand the structure of a data pipeline model equally correct when implemented in Jayvee compared to Python/Pandas."*. We therefore chose a two-sided Wilcoxon signed-rank test, with the results shown in Table 3.

Table 3: Wilcoxon signed-rank test for $H_{0,2}$: $correctness(JV) = correctness(PY)$

n	Mdn_{JV}	Mdn_{PY}	W-val	alternative	p-val	RBC	CLES
57	1.0	.92	183	two-sided	.002*	.55	.67

* $p \leq .05$

We have reason to reject the null hypothesis and instead adopt $H_{1,2}$: *"Non-professional programmers can understand the structure of a data pipeline model not equally correct when implemented in Jayvee compared to Python/Pandas."*. The CLES indicates a medium effect size. From the distribution shown in Figure 4 it is clear that participants achieved significantly higher correctness when completing the experiment using Jayvee code compared to Python/Pandas. We consider this result of practical relevance because a large improvement of correctness when interpreting data pipelines will lead to significant reduced errors when working with them.

4.3 Descriptive Survey

The follow-up descriptive survey was filled out by 56 participants. Their impressions of difficulty for understanding the data pipelines in Jayvee and Python/Pandas were answered on a 5-point Likert scale. The exact distribution of the answers can be found in Figure 10 (section 7).

After calculating medians as described in subsection 3.3, we again chose the non-parametric Wilcoxon signed-rank test because the data is paired and the differences in ordinal data from Likert scales can be ranked (Wohlin et al., 2012). The null hypothesis we defined for speed was $H_{0,3}$: *"Non-professional programmers do not perceive a data pipeline model as easier or harder to understand when implemented in Jayvee compared to Python/Pandas."*, we therefore chose a two-sided test, with the results shown in Table 4.

Table 4: Wilcoxon signed-rank test for perceived difficulty of using Jayvee compared to Python/Pandas, $H_{0,3}$: $difficulty(JV) = difficulty(PY)$.

n	Mdn_{JV}	Mdn_{PY}	W-val	alternative	p-val	RBC	CLES
56	2.0	2.0	380.5	two-sided	.153	-.23	.41

We have no reason to reject the null hypothesis and adopt $H_{0,3}$: *"Non-professional programmers do not perceive a data pipeline model as easier or harder to understand when implemented in Jayvee compared to Python/Pandas."*

4.4 Qualitative Survey Responses

In order to identify reasons for the observed effects to answer RQ2: *What reasons exist for effects on bottom-up program comprehension for data pipelines implemented in Jayvee compared to Python/Pandas for non-professional programmers?*, we used thematic analysis according to Braun and Clarke (2012).

To complement the quantitative data analysis of experiment results in our mixed-methods design, we collected qualitative responses to describe causal effects that might have influenced participants' task performance to open up future research directions and new hypotheses to explore. Our goal was to capture the diversity of effects that participants described rather than make additional statistical claims, so we included any relevant insight.

As described in section 3, we worked iteratively and tracked code assignments as well as codebook changes and chose theoretical saturation to judge the maturity of our theory (Bowen, 2008). Figure 5 shows the cumulative sum of code assignments compared to codebook changes during the thematic analysis, with every iteration highlighted by a vertical red line.

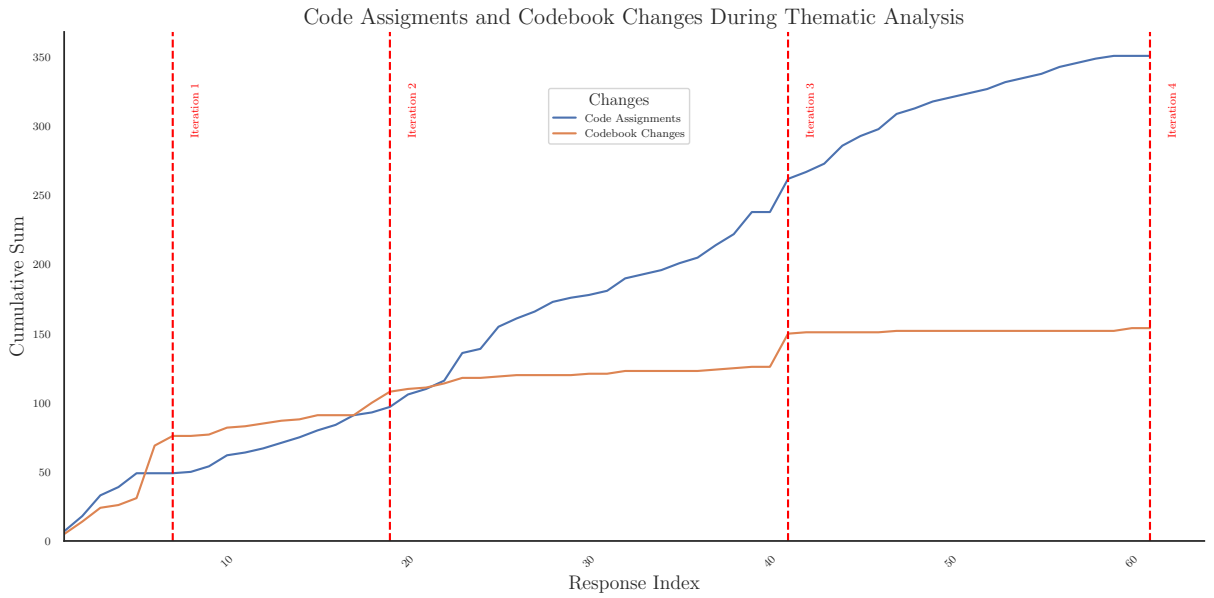


Figure 5: Code assignments compared to codebook changes during thematic analysis, showing codebook changes being rare after the third iteration, while codes were consistently applied to new responses.

We measured inter-rater reliability using Cohen's Kappa κ by two authors using the codebook to code new responses after every iteration. While κ fluctuated due to the rising complexity of the codebook and the increasing number of

codes, it consistently showed "substantial" agreement between the coding authors ($\kappa_1 = .79$, $\kappa_2 = .74$, $\kappa_3 = .64$, $\kappa_4 = .68$) (Landis and Koch, 1977).

While codebook changes are frequent initially, they become much less frequent after the third iteration. Note that the high amount of codebook changes directly before the end of an iteration is due to the adaptations that are made after the qualitative discussion by the authors after coding a subset of responses. With changes being very rare during the fourth iteration, we considered theoretical saturation to be reached and are confident our codebook encapsulates the content of the survey responses well.

We present the results of our thematic analysis according to Braun and Clarke (2012) as a collection of themes with thick descriptions. Beyond the themes that directly relate to the research questions, we also gained further insights on the role of documentation and language ecosystems. However, here we include the subset of themes that directly relate to the results from the controlled experiment. Please refer to the replication package for the full codebook with all themes and extended descriptions of codes, including additional quotes from participants [3].

Figure 6 shows the themes that emerged from coding, with six themes related to the programming language and three themes involving human factors.

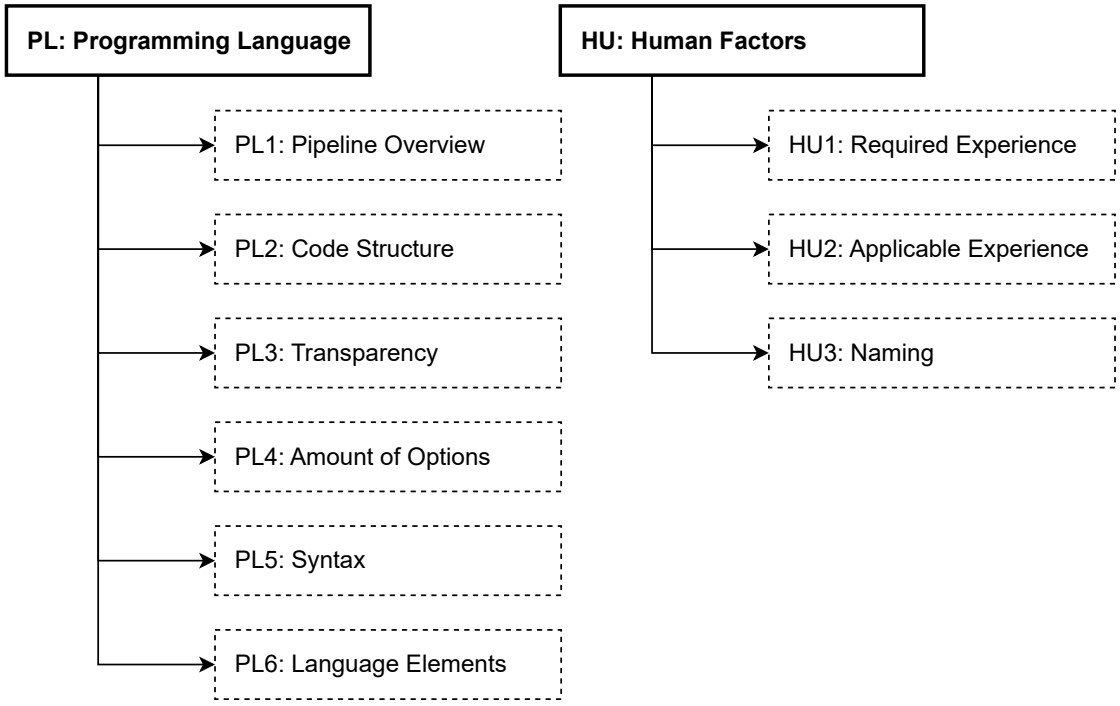


Figure 6: Overview of the codebook with two categories of themes, one related to the programming language directly and additional human factors.

In the rest of the chapter, we describe the themes in detail and highlight representative quotes from the surveys to give a vivid impression of the major topics in each theme.

4.4.1 PL1: Pipeline Overview:

Jayvee splits block definitions and the wiring-up of a pipeline by connecting blocks into separate code locations (in the example Jayvee model Listing 1, block definitions start in line 6 while the overview is created in lines 2-4). This provides an overview of the pipeline without showing any implementation details apart from the block name.

In contrast to Jayvee with its strictly enforced structure, this overview does not always exist in procedural Python scripts that are executed from top to bottom, such as the data pipelines in the experiment. The use of Pandas does also not enforce such a structure.

A major effect of this overview is that participants **can ignore code that is not immediately needed to understand the data pipeline**. This in turn improves speed for a high-level understanding because less code has to be read as described by S18: "The pipeline gives a very quick overview over what happens. When the blocks are named clearly everything can be seen on one quick view."

However, if an in-depth understanding of the implementation details is actually important to understand the data pipeline, the effect of a centralized overview on speed and understanding can potentially be negative. A few participants described a negative effect on both speed and understanding due to the additional navigation needed to read all source code. For example, S40 answered: "(Jayvee is difficult to understand...) due to the code structure/layout, need to go back & forth to search for the specific function."

The centralized overview **improved understanding of data flow and order of execution**. Especially in the domain of data engineering, the combination of being able to know how the underlying data that is manipulated by a program is changed as well as in what order source code is executed is important. For example, S37 wrote, "(...) since we have a syntax that very well shows the actual flow of the pipeline (via the block -> block -> ... syntax), it also easily understandable what blocks are executed in which order."

Summary: A data pipeline overview can be separated from implementation details in source code. The enforced structure of Jayvee means this overview always exists, while this is not true for Python/Pandas.

- *Ignoring not needed code* improves speed and understanding. However, additional navigation can mean the effect becomes negative if reading details are required.
- *The existing overview* improves understanding of data flow and order of execution.

4.4.2 PL2: Code Structure

Code structure refers to both the way source code is structured, as well as the amount of structure that is enforced by the language. The most significant difference in the way code is structured is the use of the pipes and filters architecture, with connected blocks in Jayvee compared to the script-style implementation in Python/Pandas.

Regarding the amount of enforced structure, Jayvee is much stricter than Python/Pandas. As a general-purpose programming language, Python must allow for more flexibility to enable developers to implement a wider range of programs. In contrast, as a domain-specific language, Jayvee can enforce a structure that is very close to the domain of data pipelines.

This **consistently enforced structure** enables most survey participants to understand Jayvee better, e.g., S29: "Big difference is the structure which Jayvee kind of enforces and developer can easily recognize." The improved recognition of the structure due to how consistently it is applied is a major element of the positive effect on understanding.

The **use of blocks to structure data pipeline code** is highlighted as a positive influence on pipeline understanding, especially for non-professional programmers. For example, S8 likens the experience of using blocks to using LEGO: "The best part in Jayvee is block type coding, it is similar to LEGO and you can easily remember, read and write your code."

Of course, a similar code structure can be achieved using Python with functions or classes but the increased flexibility means that it is not enforced and often not done as S26 points out: "The concept of blocks: You can manually create this in Python, but hardly anybody will do this."

Lastly, **the encapsulation of related code** is described by participants as making it easier to understand the data pipeline. S44 writes: "Jayvee is much easier to understand because every step is divided into blocks the block types are very easy to understand. A single operation is performed in one block, which makes it easy to comprehend." Importantly, encapsulated code must be sliced so that only a single operation is done in one unit, or participants consider it a detractor for understanding.

Summary: Code structure refers to the way source code is organized. Different languages enforce a more or less consistent structure.

- *Stricter enforcement of structure* improves understanding and increases learning effects from other data pipelines.
- *Consistent structure* allows readers to quickly find expected elements, such as the data pipeline overview.
- *Using blocks* is a positive influence on pipeline understanding and aligns with the mental model of data pipelines.
- *Encapsulation of related code* makes it easier to understand data pipelines, as long as a single operation is performed in each section.

4.4.3 PL3: Transparency

Transparency relates to how deeply participants can understand the operations performed in the data pipeline by just reading the source code. Differences can come from how visible implementation details are, depending on the level of abstraction a language aims for. Additionally, how much functionality can be expressed in few lines of code (which we call density of functionality) affects transparency in the sense that with high density of functionality less low-level operations are expressed in source code.

Python/Pandas was identified as having a much higher **density of functionality** than Jayvee. Regarding the effects, participants had mixed impressions. On one side, being able to express a lot of logic in a few lines of code makes each individual line of code harder to understand, potentially decreasing correctness as S30 explains: "Python makes it possible to have a lot of functionality in just a few lines, which can make it hard to read if you have not written it yourself."

The tradeoff is that pipeline models in a less expressive language must consist of more source code which is slower to read. S0 mentions this concern: "Especially in a large pipeline a file might get really big because of all the definitions (especially unnecessary empty block definitions).". However, because the data pipeline models in our experiment were comparatively small, the majority of participants did not describe this problem.

One way to achieve a high density of functionality is to implement a high **degree of automatic decisions** and many operations in one unit of code. As an example, loading data with `read_csv()` can use various sources and automatically chooses structure and data types based on the underlying data that cannot be inferred from the source code alone. Additionally, the structure of the output can potentially change without any change in the source code if the input data changes.

Increased automation by grouping many operations in one unit of code makes data pipelines harder to understand and decreases correctness. Often, library methods of Pandas are singled out by participants for this kind of complexity, with S0 remarking: "Difficult: The methods sometimes do many things at once (example: load to a sqlite file and automatically choose data types)." S26 describes a similar experience: "Functions like `pd.read_csv` are hard to understand, as they can read a DF from so many sources (in Jayvee you have one datasource specified)."

Instead of increased automation, the **inability to see all implementation details** was identified as a negative effect on the ability to understand the data pipeline by participants. This effect was mostly found in Jayvee, with examples including the `TableTransformer` block that takes input columns and output columns as properties, for which participants were unsure if it keeps or removes the input columns.

Summary: Transparency relates to how well participants can understand every operation performed in a data pipeline based on the source code alone.

- *High density of functionality*, many operations per line of code, is a challenge to understanding for small data pipelines. However, reading larger data pipelines will be slow and potentially error-prone with lower density of functionality.
- *Increased automation* makes data pipelines harder to understand and decreases comprehension correctness.
- *Hidden implementation details* can negatively affect the understanding of data pipelines.

4.4.4 PL4: Amount of Options

A common theme in the survey responses was the large number of options to implement functionality in Python/Pandas and the comparatively few options in Jayvee. For example, to download a CSV file, Python programmers could use the standard library with `urllib` or use Pandas `read_csv()` with nearly equivalent outcomes. DSLs can focus on a few core features and only provide one solutions for these.

The effect of **many competing options** was described as a detriment to understanding by participants such as S49: "In Python, there are many varieties and different options, libraries etc, it is harder for non-experienced to grasp the essence." As they describe, these challenges impact mostly non-professional programmers or programmers unfamiliar with the language itself.

External libraries exacerbate this effect, adding additional ways to solve problems with potentially multiple libraries that solve the same set of problems. Moreover, every library has its own mental model of the problem space with their own glossary, code styles and documentation. S0 writes: "In Jayvee everything (all blocks) are from the same source, while in Python there are many libraries with different method styles and documentation."

External libraries also evolve independently of the main language and each other. This means developers must keep up with changes from different sources to keep their understanding of source code up-to-date, or risk interpreting new library code wrongly.

Despite the challenges that external libraries introduce, their availability has obvious upsides, e.g., less work to implement common functionality. Managing the scope of language features and how external libraries are used is therefore a tradeoff that depends on the experience level of the main users of the language.

Summary: The amount of options to implement the same functionality varies greatly between languages, with GPLs having to be more flexible than DSLs. External libraries add additional approaches.

- *Many competing options* to solving the same problem are a challenge to understanding data pipelines, mainly for less experienced readers.
- *External libraries* increase the amount of available options and have different mental models and glossaries. However, aside from their negative effect on understanding, external libraries reduce required work to implement data pipelines.

4.4.5 PL5: Syntax

Participants sometimes commented on the syntax differences of the languages as reasons for their performance. Both languages were described as **human-readable**, sometimes as being like English text or pseudocode. Human-like language syntax was generally linked to making it easier to understand the data pipeline, e.g., by S31: "Jayvee has a very human-like language, almost like pseudocode which can be immediately understood even by non programmers in my opinion as long as they have a basic theoretic knowledge about pipelines."

While Python is well known for its closeness to pseudocode, Jayvee uses considerably more special characters and an uncommon structure. We attribute the positive comments on Jayvee's human-like syntax largely to the use of a glossary that is close to the problem domain, e.g., the use of domain entities such as `pipeline` as part of the syntax. Reusing a glossary that is familiar to domain experts allows them to more easily understand the meaning of data pipeline code.

In contrast, encountering **unfamiliar syntax** is described as a challenge to understanding data pipelines from code. This was mostly an issue for participants solving tasks in Jayvee as they had less previous experience with the language. However, some participants described similar problems with the syntax used by libraries in Python, for example, Pandas creating new columns in a Dataframe with an assignment operator instead of a function call.

Summary: Language syntax is discussed by participants, but largely in regard to personal preference for more familiar languages like Python.

- *Human-readable* syntax makes it easy to understand a data pipeline. Both Python and Jayvee are described as human-readable languages.
- *Unfamiliar syntax* has a negative effect on understanding. New languages and unfamiliar external libraries can introduce this effect.

4.4.6 PL6: Language Elements

Language elements have a large influence on understanding of data pipeline code. GPLs such as Python must by necessity also provide general-purpose language elements, such as classes or functions, that can be used to build systems for any use case. In contrast, DSLs can express domain concepts such as pipelines, blocks and pipes, or value types directly as language elements.

The **use of domain-specific language elements** is described as making it easier to understand the data pipeline by participants. The explicit blocks and pipes structure that is enforced by Jayvee aligns closely with how users visualize data pipelines. Readers can then directly build their mental model of the data pipeline from the similar representation in the source code.

Other language elements negatively impacted understanding with some participants mentioning that **Jayvee language elements are unusual** and need to be learned (in contrast to Pythons language elements that are largely known from other GPLs).

An example are value types based on constraints, as S51 points out: "I found the Jayvee code structure a bit difficult to understand, mostly the constraints and value type." A possible explanation could be that value types and constraints align less obviously than blocks and pipes with the visual model of a data pipeline.

For Python, the **use of advanced programming concepts** was mentioned as a problem participants faced understanding the experiment tasks. Concrete examples are described by S12: "Some functions like lambda, list comprehension and implicit operations are not intuitive and require documentation and comments to understand." Advanced programming elements have to be used carefully and sparingly if the goal is to create a data pipeline that can be understood by relative junior programmers.

Summary: Python must provide general-purpose language elements such as classes and functions, while DSLs can introduce domain concepts such as pipes and blocks.

- *Using blocks as domain-specific language elements* improves pipeline understanding and is intuitive because it aligns with the visual model of a data pipeline.
- *Unusual language elements* such as value types based on constraints are a challenge to pipeline understanding.
- *Advanced programming concepts* like lambdas or list comprehension make pipeline understanding harder, especially for programmers without previous experience in the language.

4.4.7 HU1: Required Experience

Understanding data pipeline code is influenced by the previous experience of the reader. Depending on the tool used to implement the data pipeline, more or less experience might be needed. Further, the type of experience also matters. Subject-matter experts are often experts in the data they are working with, but might not have extensive software engineering experience.

The **need for previous experience with programming** to understand Python/Pandas code is mentioned by multiple participants in their surveys. As a GPL, Python must have many features and allow for a maximum amount of flexibility, which makes it inherently complex. Furthermore, more knowledge of programming is involved because the concepts expressed in the language cannot be domain-specific but have to be generic (e.g., classes and functions). S34 expresses the difference: "I think the difference might have mostly to do with how much experience one has in programming; I think that Python might require quite some knowledge to get used to, while Jayvee is a bit easier to understand even as a person with not much programming experience."

The more flexible a language is, the more experience and discipline is needed to stick to good practices and write code that is easy to understand. With the ease of writing script-style Python code, it is not uncommon for developers to implement prototypes in Python that later on get promoted to production code without a rewrite, creating hard to understand data pipelines.

Summary: Required experience refers to the amount of experience required to understand a data pipeline from source code. For reading source code, the main required experience is previous programming.

- *Previous experience with programming* is needed to understand Python because of the use of generic programming concepts. In contrast, Jayvee is easier to understand for non-programmers because it is using domain-specific concepts.
- *More flexibility* means more experience is needed to follow good habits and make code easily readable.

4.4.8 HU2: Applicable Experience

How closely a language aligns with the mental model of data pipelines is important to reuse experience outside of software engineering. Participants describe Jayvee's blocks and pipes structure as intuitive because it mirrors how they think about data pipelines. This positively affects understanding, e.g., S35 explains why Jayvee pipelines are easy to understand: "Jayvee code steps are directly mapped to the data engineering pipeline lifecycle."

However, the close match to the mental model must be carefully maintained; otherwise it can lead to confusions. One such mismatch were the interpretation blocks in Jayvee (such as the `TextFileInterpreter`) to convert binary data to text data. Participants were confused about what the interpretation blocks did because the level of abstraction was lower than what they expected.

A special case of applicable experience is building up knowledge from previous experience with the same tool. **High flexibility means even similar pipelines can look very different**. A challenge with the low enforced structure of Python/Pandas is that learning effects from creating or reading other data pipelines are reduced. S29 summarizes the challenge as "No structure, every pipeline is a new pipeline." This effect is worsened by the amount of different libraries that can be used to solve common problems, meaning experienced in one library does not necessarily apply to data pipelines that use a different library.

Summary: Being able to reuse experience from other sources, such as working with spreadsheets, means data pipelines can be understood by a wider range of readers. Often, subject-matter experts might lack programming experience but have previous domain experience.

- *Alignment of code to the mental model of data pipelines* improves understanding, even without programming experience. However, creating the expected abstraction level is important or readers are confused.
- *Learning effects are reduced* when similar pipelines can look different in source code due to high flexibility.

4.4.9 HU3: Naming

Good names improve understanding, especially for non-professionals. However, as Phil Karlton said "There are only two hard things in Computer Science: cache invalidation and naming things." [4]

Generally, participants describe names in Jayvee as easy to understand, probably because they are close to the terminology of the domain of data pipelines. In contrast, survey answers mention Python and Pandas as having inconsistent and sometimes confusing naming, potentially because of the generality required by being a GPL and due to the use of external libraries with an inconsistent glossary.

Well named processing steps, both for language elements and user-defined names, have multiple positive effects. Speed is improved by being able to skim source code and clear names make it easier to understand the data pipeline as a whole, S18 writes: "When the blocks are named clearly everything can be seen on one quick view. That makes the pipeline easier to understand."

Good names must **follow a consistent approach**, which in turn improves understanding. This is a challenge for a GPL like Python because much of the domain-specific functionality comes from external libraries such as Pandas that have different glossaries and approaches to capturing the domain.

Lastly, under the assumption that names are chosen well, the **quantity of naming opportunities** is important as well, with a higher quantity of names making it easier to understand a data pipeline. Script-style data pipeline implementation give few opportunities for good naming of steps, meaning developers must resort to comments if they want

[4] https://martinfowler.com/bliki/TwoHardThings.html

to communicate reasoning. Due to named blocks, Jayvee provides more naming opportunities, both for language elements and user provided names that explain the intent behind the use of a block.

Summary: Naming of elements in a pipeline has a major effect on how easy the resulting source code is to understand.

- *Good names* improve understanding by allowing readers to skim the source code and get an overview of the whole pipeline.
- *Consistent naming* has a positive effect on understanding. External libraries with their own glossary can make naming less consistent.
- *The quantity of human-provided names* is important to communicate intend, with a positive effect on understanding if the names are chosen well.

5 Discussion

Based on the results, a DSL based on a pipes and filters structure can be a valuable tool to build data pipelines with subject-matter experts. Participants with a non-professional programmer background can understand data pipeline source code more correctly, but not faster or more easily.

A possible explanation for the similar speed is that the participants had considerably more previous experience with Python/Pandas than with Jayvee, which likely influenced how fast they were able to understand the data pipelines in favor of Python/Pandas. This will not be an uncommon situation however, because a new DSL always presents a learning challenge, while many practitioners might already have worked with Python and Pandas. However, the fact that participants were still able to complete the tasks with Jayvee in a similar time indicates that learning a new DSL can be done in limited time and provide other benefits like improved correctness, even for non-professional programmers.

Additionally, Jayvee is considerably more verbose than Python/Pandas, and therefore took participants longer to read before they could solve the tasks. In the context of open data, the tasks were representative of real-life challenges and based on real open data sets. Most open data sets are small, mostly under 10 MB and published in tabular formats such as CSV (Umbrich et al., 2015; Mitlohner et al., 2016). However, for larger scale data pipelines, e.g. in industrial settings a more expressive syntax is needed. For these situations, we expect that the difference in speed for program understanding would increase in favor of Python/Pandas due to Jayvee's verbosity and structure.

Similarly, more complex tasks could require functionality outside the limited feature set of Jayvee. In previous studies, we have found that in these situations perceived implementation difficulty increases sharply, and it stands to reason that program understanding would decrease as well (Heltweg et al., 2025).

During the experiment, both Jayvee and Python/Pandas source code was displayed as text, without syntax highlighting or the use of an IDE. We chose to not provide an IDE because the maturity of tool support for Python/Pandas and Jayvee differs significantly and would have introduced a confounding factor. In similar work, replication studies of experiments with the addition of IDE support have shown that correctness improves for all treatments, but the relative differences between them remain consistent (Kosar et al., 2018). Therefore, we expect that the results of our experiment would not change significantly with the addition of IDE support.

The code structure of the Python/Pandas data pipelines might have an effect on the results. We chose to use script-style implementations in Python with Pandas, as they are common in practice for smaller data pipelines As discussed in subsubsection 4.4.2, classes and functions can be used in Python to create a structure similar to Jayvee which would reduce the effects of using a DSL.

With regard to task design, we chose to focus on comprehension tasks of data pipeline structure as a first step. Alternative task goals, such as locating errors or predicting the output of a data pipeline could be used in future work. We consider the comprehension of data pipeline structure as a necessary prerequisite for these tasks. From the qualitative feedback, we expect that the results would be similar for correctness, with Jayvee being more verbose and prescriptive with less functionality. Especially the exact structure of data pipeline output was often unclear to participants due to the automated Dataframe structure creation when loading a data set with Pandas.

Of course, program understanding is only one part of the software development process and other tasks such as extending existing programs or code creation would likely show very different results. We expect implementations in Jayvee to be slower due to the increased verbosity and more strict structure, but additional studies are needed to verify these assumptions.

5.1 Learnings for Language Designers

Multiple design decisions are contributing factors to the improved performance and can provide guidelines for future developers of DSLs.

Representing a data pipeline with blocks and pipes as first class language elements seems to be a good choice. It is described as intuitive and clear, especially because it clearly aligns with the mental model of data pipelines as the reader visualizes them.

A data pipeline overview that is represented directly in the syntax of the source code and separated from the implementation details is consistently highlighted as an important positive influence. In addition, the strongly enforced structure of a data pipeline program means readers can quickly orient themselves in the source code and learn with every pipeline they read.

The effect of well-named language elements was considerable, indicating that names are a major influence on data pipeline understanding and especially to provide context to implementation decisions. Consequently, language designers should pay attention to not only using a consistent glossary to name language elements, but also to providing opportunities for developers to use many descriptive names. As an example, by encapsulating functionality into named blocks, data pipelines implemented in Jayvee have a greater minimum amount of named elements than script-style implementations in Python/Pandas. Because this structure is strict, even non-professional programmers are guided to describe the steps they implement in any given pipeline.

Regarding complexity, providing multiple options that achieve the same goal, both in syntax as well in approaches to solve a problem, has been discussed as a barrier to understanding by participants. Because of this, introducing additional syntax or syntactic sugar to make one specific use-case easier should always be seen as a tradeoff between the expressiveness of the language versus the added complexity.

6 Limitations

As a mixed-method study, multiple sets of limitations are potentially relevant to correctly evaluate the results. We evaluate limitations and ways to mitigate them in regard to the quantitative data from the subsection 3.2 and the survey questions, based on threats to validity described in Wohlin et al. (2012). Trustworthiness criteria according to Guba (1981) are used for the follow-up qualitative work with answers from the descriptive survey (subsection 3.3).

While we present more than one set of limitations in this chapter, it is important to highlight that the mixed-method approach of this study (with data- and method-triangulation) allows the individual methods to partially make up for the weaknesses of the other. This means the overall research design contributes as a mitigating factor for some of the discussed limitations.

6.1 Threats to Validity

We describe potential threads to validity according to the framework presented in Wohlin et al. (2012).

Conclusion Validity

Threats to conclusion validity are challenges to understanding the correct relationships between the treatment and results of an experiment.

The DSL that was investigated as treatment is in large parts designed and implemented by the authors of this study, therefor bias and searching for positive results is a clear threat to conclusion validity. In an attempt to reduce its impact, we defined the research design as well as hypotheses to analyze ahead of data collection, based on indicators found in previous work (Heltweg et al., 2025) and used standard research designs and statistical tests. Additionally, we reported effect sizes and the results of all hypotheses tests, including ones without statistically significant results such as time spent on task. During data collection, we followed an experiment procedure document to reduce the introduction of individual bias when guiding participants through the experiment. In addition, participants purely interacted with an automated experiment tool that implemented the treatment and took measurements impartially without interaction by the researchers. Nonetheless, subconscious bias remains as a threat to conclusion validity. Therefore, we have shared the experiment tool[3] to allow for thorough review and independent replication.

Normally, the heterogeneity of students as participants also provides a challenge. However, the use of a crossover experiment design mitigates this concern because they measure differences in comparison to the participants' average and not between participant groups (Vegas et al., 2016).

Internal Validity

Internal validity describes the extent to which influences outside the control of the researcher, apart from the treatment, influence the results of the experiment.

If the tools or tasks used for the experiment were of low quality, they could introduce external factors to the results. In order to reduce these influences, we tested the tool and task implementations in multiple sandbox tests with other researchers and in pilot experiments with individual students from earlier semesters and adjusted them based on feedback, as suggested by Ko et al. (2015).

Before the experiment runs, one of two researchers explained the experiment procedure to participants and answered questions. Differences in communication style could introduce a threat to internal validity. We mitigated this by preparing an experiment procedure document that was followed by both researchers. In addition, due to the crossover design, every experiment cohort that was instructed by one researcher completed tasks with both treatments and the experiment results depend on the delta in their individual performance, not between groups. Nonetheless, the use of multiple researchers to instruct the participants could have influenced the results between groups.

By selecting volunteers out of a class of students, the results may be influenced if participants think positive responses in regard to Jayvee would have a positive influence on their grade. We therefor clearly communicated to students that data would be anonymized and participation or performance in the experiment would have no effect on their grade.

The differences in previous experience with Jayvee compared to Python/Pandas also introduces a threat to internal validity. We mitigated this by introducing Jayvee with two lectures and at least one practical exercise before the experiment. We also collected and reported the previous experience of participants with both languages to allow for a better contextualizing of the results at the start and end of the semester, but not directly before the experiment. It is likely that the differences in previous experience with the languages influenced the results, especially regarding speed and perceived difficulty. However, we consider the results interesting, because due to its popularity, data practitioners

often have previous experience in Python/Pandas and not in new DSLs. We consider our study as a first step to establish initial insights. In further work, replication studies with more balanced previous experience would be needed to confirm the results.

Crossover designs introduce the threat of carryover and familiarization effects, in which the administration of one treatment might influence others. It must be explicitly discussed as a threat to internal validity according to Vegas et al. (2016). We minimized carryover during the experiment design time in multiple ways. First, by randomly assigning participants to different treatment orders. Second, to reduce the effect of increasing familiarity with the experiment tool itself influencing later task performance, we added an initial task using pseudocode and placeholder step names before applying the real treatments. Lastly, we added a stage of hidden source code, so participants could read the available steps in the pipeline first to reduce the effect of recognizing some steps from the previous task.

Regardless of these measurements, we must recognize that carryover could still be an influencing factor on the results and aim for future replication with between-subject designs.

Construct Validity

Construct validity is concerned with the appropriateness of the experiment construct to measure the underlying concept or theory and the ability to generalize the result of the experiment to it.

The dependent variables in the experiment were clearly defined and measured programmatically. Time and correctness are the most common measures used in bottom-up code comprehension experiments (Wyrich et al., 2023). The concrete definition of correctness for a data pipeline that we used is not previously validated; however, we consider it appropriate because it covers the correct understanding of both selection and the order of steps.

Because only one measurement was taken for each construct, mono-method bias is a concern for the controlled experiment part of this study. This limitation is mitigated by the fact that additional insights about the underlying concepts are drawn from qualitative data as part of the mixed-method design. Nonetheless, additional experiments with more measurements should be done in future work to strengthen the quantitative results.

External Validity

External validity is the ability to generalize the results, e.g., to an industry context.

We chose Masters level students as proxies for a population of subject-matter experts working with data in industry, that are non-professional programmers. When drawing conclusions from the results of this study, it is important to contextualize them with this limited population in mind (Falessi et al., 2018). Using students allows us to gather more data points, establish a trend and prepare future studies with practitioners (Tichy, 2000). Additional experiments, replicating the same setup, with real subject-matter experts from industry would be needed, but we expect the results to generalize well. Other populations, such as professional programmers from industry, would very likely encounter different challenges and the results of this study should not be taken as indication for their experience.

Because we allowed students to voluntarily opt in to the experiment, only 57 of the 98 students that completed the course participated. We consider this number to be high enough to be representative of the population, however it is possible that less invested students did choose to skip the experiment.

6.2 Trustworthiness criteria

For the descriptive survey, we use the trustworthiness criteria of credibility, transferability, dependability, and confirmability (Guba, 1981).

Credibility

Our goal was to establish credibility, how well the findings represent the real effects, with various types of triangulation in the mixed-methods research design (Thurmond, 2001). By combining the quantitative data from a controlled experiment with the qualitative data of the descriptive surveys, we establish method and data triangulation. In addition, large parts of the qualitative data were coded by multiple researchers as a form of investigator triangulation.

The opt-in, voluntary nature of the experiment introduces a potential bias in the participant selection for more motivated students. We mitigated this effect but clearly stating that participation would have no effect on course grades, both verbally and in the experiment handout we provided to participants.

Transferability

Transferability, how well the results apply to other contexts, has to be discussed from multiple angles. First, the use of students as participants is problematic when attempting to generalize to professionals in industry, additional context is provided in the discussion regarding the external validity of the experiment that also applies to the qualitative part of the study.

Second, the responses of participants must be seen in the context of one specific DSL, Jayvee, and might not transfer to other DSLs. The descriptions of themes should be seen under this aspect, and additional research with different DSLs is needed to make sure the findings transfer to other languages.

Lastly, the data pipelines that participants had to understand during the experiment were relatively small (but based on real-world open data sets). How well the results transfer to larger scale data pipelines is unclear. When appropriate, we discussed the potential trade-offs regarding small and large data pipelines in the descriptions of the themes (e.g., regarding density of functionality).

To increase transferability, we provided thick descriptions of the themes and extensive quotes from participants in support (as well as an additional, extended description of the themes [3]). Future researchers can use this additional context to evaluate the research results in additional contexts.

Dependability

For dependability, making sure the findings are consistent and can be repeated, we reported the research design in detail and provided as much data as possible. In addition, the complete survey question export and code used to analyze the data is available.

Confirmability

Confirmability, how well the findings represent the objective reality and are not influenced by researcher bias, is challenged by the involvement of the authors in the implementation of Jayvee. Because this introduces a risk of bias, we took steps to introduce additional data and method triangulation by prefacing the survey with a controlled experiment with automated measurements that is less subjective to researcher bias. Regardless of the mitigations employed, we have to acknowledge our own bias and would welcome replication by neutral parties. To enable other researchers to confirm our findings, we have established an audit trail by describing the research design in detail and providing as much data used during the analysis as possible. Thick descriptions of the themes and direct quotes from the survey also give additional context to the findings.

7 Conclusion

In this mixed-methods study, we have asked two research questions: First, *do data pipelines implemented in Jayvee change bottom-up program structure comprehension compared to Python/Pandas for non-professional programmers regarding speed, correctness and perceived difficulty?* and second, *what reasons exist for effects on bottom-up program comprehension for data pipelines implemented in Jayvee compared to Python/Pandas for non-professional programmers?*

To do so, we have executed a controlled experiment with 57 volunteers students comparing their performance on data pipeline understanding tasks implemented in Jayvee and Python with Pandas. In addition, participants could provide qualitative feedback in a post-experiment survey that we then analyzed using qualitative data analysis.

Based on the experiment data, participants are neither faster, nor consider it easier to understand a data pipeline implemented in Jayvee compared to Python/Pandas (Figure 9, Table 2). However, participants can understand a data pipeline significantly more correctly (Figure 4, Table 3).

Qualitative analysis of participant feedback revealed a variety of possible reasons for these effects, summarized in Figure 6. Data pipelines in the experiment were based on real-life open data sets, but relatively small and further studies would be needed to verify that these effects generalize to larger and more complex data pipelines.

Predictably, most effects are grounded in the difference between programming languages themselves. Participants highlight the pipeline overview provided by Jayvee as a major positive influence on understandability. This overview is enforced due to the more rigid structure of Jayvee programs that make them easier to understand than Python/Pandas scripts. How deeply participants could understand the data pipeline, the transparency of source code, had mixed effects, with high density of functionality and increasing automation making a pipeline harder to understand but faster to read. Similarly, the amount of available options, especially with the introduction of external libraries, is a challenge to understandability but reduces the work needed to implement pipelines in the first place. Unfamiliar syntax was an additional problem for some participants, even if both Jayvee and Python were described as human-like languages. Lastly, provided language elements are a factor in the different outcomes because, as a domain-specific language, Jayvee could include language elements that were intuitive to understand in a data pipeline context while some participants struggled with advanced programming concepts like lambdas in Python.

In addition to the effects of the programming languages themselves, we also identified several human effects. First, the previous experience required to understand data pipelines from source code differs between the approaches. Participants identify previous programming experience as a necessary precursor to understanding data pipelines written in Python/Pandas, while they consider pipelines written in Jayvee to be approachable by novices. Second, the implementation language effects which previous experience is applicable to understanding a data pipeline. If the abstraction level is maintained well, a domain-specific language like Jayvee allows readers to reuse previous experience from data engineering with other tools like visual modeling software. Finally, depending on the reader, well-chosen, descriptive names have a large influence on how understandable data pipeline source code is. Languages with a wide library ecosystem like Python with Pandas face challenges to keep a consistent glossary between different authors. Additionally, the strict structure of Jayvee with extensive possibilities for user-provided names allowed future readers to infer additional information.

Besides the effects that are often described and have a clear influence, open questions remain. For example, the best abstraction level of a domain-specific language for data pipelines is unclear and might depend on the intended audience. Additionally, a good tradeoff between the reuse of work with a library ecosystem versus the complexity it introduces warrants further studies. Density of functionality shows a similar tradeoff between short to write and expressive code versus harder to understand pipelines. With more research, it might be possible to identify the reasons for the largest negative effects and avoid them in future language design.

In summary, domain-specific languages such as Jayvee have the potential to be more correct in the domain of data pipeline modeling. These effects are especially strong for non-professional programmers, such as subject-matter experts in other domains. A variety of reasons for these effects exists, largely based on the programming language itself or on the type of reader that tries to understand the source code. However, the exact effect of many reasons is still an open question that needs further research to develop a comprehensive theory of domain-specific languages for data pipeline modeling.

In future work, we intend to explore more narrow features of domain-specific languages for data engineering, such as value types or selection syntax for tabular data, with additional controlled experiments.

Acknowledgements

This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17045 Software Campus 2.0 (Friedrich-Alexander-Universität Erlangen-Nürnberg) as part of the Software Campus project 'JValue-OCDE-Case1'. Responsibility for the content of this publication lies with the authors.

Data Availability Statement

The data generated and analyzed during the current study is available on Zenodo at:
`https://doi.org/10.5281/zenodo.15574873`.
For convenience, the full codebook is also hosted at:
`https://rhazn.github.io/2025-data-release-program-comprehension-jayvee/`.

References

Barišić, A., Amaral, V., and Goulão, M. (2018). Usability driven DSL development with USE-ME. *Computer languages, systems & structures*, 51:118–157.

Basili, V. R. and Rombach, H. D. (1988). The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773.

Bowen, G. A. (2008). Naturalistic inquiry and the saturation concept: a research note. *Qualitative research: QR*, 8(1):137–152.

Braun, V. and Clarke, V. (2012). Thematic analysis. In *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological*, pages 57–71. American Psychological Association, Washington.

Buse, R. P. L., Sadowski, C., and Weimer, W. (2011). Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA. ACM.

Cingolani, P., Sladek, R., and Blanchette, M. (2015). BigDataScript: a scripting language for data pipelines. *Bioinformatics (Oxford, England)*, 31(1):10–16.

do Nascimento, L. M., Viana, D. L., Neto, P. A. S., Martins, D. A., Garcia, V. C., and Meira, S. R. (2012). A systematic mapping study on domain-specific languages. In *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, pages 179–187.

Falessi, D., Juristo, N., Wohlin, C., Turhan, B., Münch, J., Jedlitschka, A., and Oivo, M. (2018). Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1):452–489.

Feigenspan, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S. (2012). Measuring programming experience. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 73–82. Ieee.

Fonseca, N., Paulo Fernandes, J., Pires, M., and Melo de Sousa, S. (2020). PACE: A DSL-based approach to manage complex build pipelines. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 43–50. IEEE.

Garlan, D. and Shaw, M. (1993). AN INTRODUCTION TO SOFTWARE ARCHITECTURE. In *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, pages 1–39. WORLD SCIENTIFIC.

Guba, E. G. (1981). Criteria for assessing the trustworthiness of naturalistic inquiries. *ECTJ*, 29(2):75.

Heiberger, R. and Robbins, N. (2014). Design of diverging stacked bar charts for likert scales and other applications. *Journal of statistical software*, 57:1–32.

Heltweg, P. and Riehle, D. (2023). A systematic analysis of problems in open collaborative data engineering. *Trans. Soc. Comput.*, 6(3-4):1–30.

Heltweg, P., Schwarz, G.-D., Dirk, R., and Felix, Q. (2025). An empirical study on the effects of jayvee, a domain-specific language for data engineering, on understanding data pipeline architectures. *Software: Practice & Experience*.

Hoffmann, B., Urquhart, N., Chalmers, K., and Guckert, M. (2022). An empirical evaluation of a novel domain-specific language - modelling vehicle routing problems with athos. *Empirical Software Engineer*, 27(7):180.

Jedlitschka, A. and Pfahl, D. (2005). Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.*, page 10 pp. Ieee.

Johanson, A. N. and Hasselbring, W. (2017). Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, 22:2206–2236.

Johnson, R. B., Onwuegbuzie, A. J., and Turner, L. A. (2007). Toward a definition of mixed methods research. *Journal of mixed methods research*, 1(2):112–133.

Kerby, D. S. (2014). The simple difference formula: An approach to teaching nonparametric correlation. *Comprehensive psychology*, 3:11.IT.3.1.

Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., and Pohthong, A. (2017). Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630.

Kitchenham, B. A. and Pfleeger, S. L. (2008). Personal opinion surveys. In Shull, F., Singer, J., and Sjøberg, D. I. K., editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer London, London.

Klanten, K., Hanenberg, S., Gries, S., and Gruhn, V. (2024). Readability of domain-specific languages: A controlled experiment comparing (declarative) inference rules with (imperative) java source code in programming language design. In *Proceedings of the 19th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications.

Ko, A. J., LaToza, T. D., and Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141.

Kosar, T., Gaberc, S., Carver, J. C., and Mernik, M. (2018). Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23(5):2734–2763.

Kosar, T., Mernik, M., and Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17:276–304.

Kosar, T., Oliveira, N., Mernik, M., João, M., Pereira, M., Repinåek, M., Cruz, D., and Rangel Henriques, P. (2010). Comparing general-purpose domain specific languages: empirical study. *Computer Science Information Systems*.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174.

Lopes, J., Bernardino, M., Basso, F., and Rodrigues, E. (2021). Textual-based DSL for conceptual database modeling: A controlled experiment. In *Anais do XXXVI Simpósio Brasileiro de Banco de Dados (SBBD 2021)*, pages 169–180. Sociedade Brasileira de Computação - SBC.

McGraw, K. O. and Wong, S. P. (1992). A common language effect size statistic. *Psychological bulletin*, 111(2):361–365.

Misale, C. (2017). *PiCo: A Domain-Specific Language for Data Analytics Pipelines*. PhD thesis, University of Torino, Italy.

Mitlohner, J., Neumaier, S., Umbrich, J., and Polleres, A. (2016). Characteristics of open data CSV files. In *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE.

Robbins, N. B., Heiberger, R. M., and Others (2011). Plotting likert and other rating scales. In *Proceedings of the 2011 joint statistical meeting*, volume 1.

Roberto Minelli, A. M. and Lanza, M. (2015). I know what you did last summer. In *2015 IEEE 23rd International Conference on Program Comprehension*, volume 3, pages 1–28.

Shapiro, S. S. and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611.

Shaw, M. and Garlan, D. (1995). Formulations and formalisms in software architecture. In van Leeuwen, J., editor, *Computer Science Today: Recent Trends and Developments*, pages 307–323. Springer Berlin Heidelberg, Berlin, Heidelberg.

Thurmond, V. A. (2001). The point of triangulation. *Journal of nursing scholarship: an official publication of Sigma Theta Tau International Honor Society of Nursing / Sigma Theta Tau*, 33(3):253–258.

Tichy, W. F. (2000). Hints for reviewing empirical work in software engineering. *Empirical Software Engineering*, 5(4):309–312.

Umbrich, J., Neumaier, S., and Polleres, A. (2015). Quality assessment and evolution of open data portals. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 404–411. IEEE.

Vallat, R. (2018). Pingouin: statistics in python. *The Journal of Open Source Software*, 3(31):1026.

Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the CL common language effect size statistics of McGraw and wong. *Journal of educational and behavioral statistics: a quarterly publication sponsored by the American Educational Research Association and the American Statistical Association*, 25(2):101–132.

Vegas, S., Apa, C., and Juristo, N. (2016). Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80.

Wile, D. (2004). Lessons learned from real DSL experiments. *Science of computer programming*, 51(3):265–290.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Science + Business Media.

Wyrich, M., Bogner, J., and Wagner, S. (2023). 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE transactions on software engineering*, 44(10):951–976.

Appendix

7.1 Task Examples

```
constraint GeographicCoordinateScale on decimal: value >=
      -90 and value <= 90;

valuetype GeographicCoordinate oftype decimal {
    constraints: [GeographicCoordinateScale];
}

block ValuetypeValidator oftype TableInterpreter {
  header: true;
  columns:[
    'uuid' oftype text,
    'latitude' oftype GeographicCoordinate,
    'longitude' oftype GeographicCoordinate,
    'bezeichnung' oftype text,
    'traeger_bezeichnung' oftype text,
    'traeger_art' oftype text,
    'website' oftype text,
  ];
}
```

```
data = data[[
  'uuid',
  'latitude',
  'longitude',
  'bezeichnung',
  'traeger_bezeichnung',
  'traeger_art',
  'website',
]]

data = data.astype({
  'uuid': str,
  'latitude': float,
  'longitude': float,
  'bezeichnung': str,
  'traeger_bezeichnung': str,
  'traeger_art': str,
  'website': str,
})

data = data[data['latitude'].apply(lambda
     input: input >= -90 and input <=
    90)]
data = data[data['longitude'].apply(
    lambda input: input >= -90 and input
     <= 90)]
```

Figure 7: Comparison of source code to filter and apply a schema to data, shown for task 2 in Jayvee and Python/Pandas.

7.2 Extended Result Data

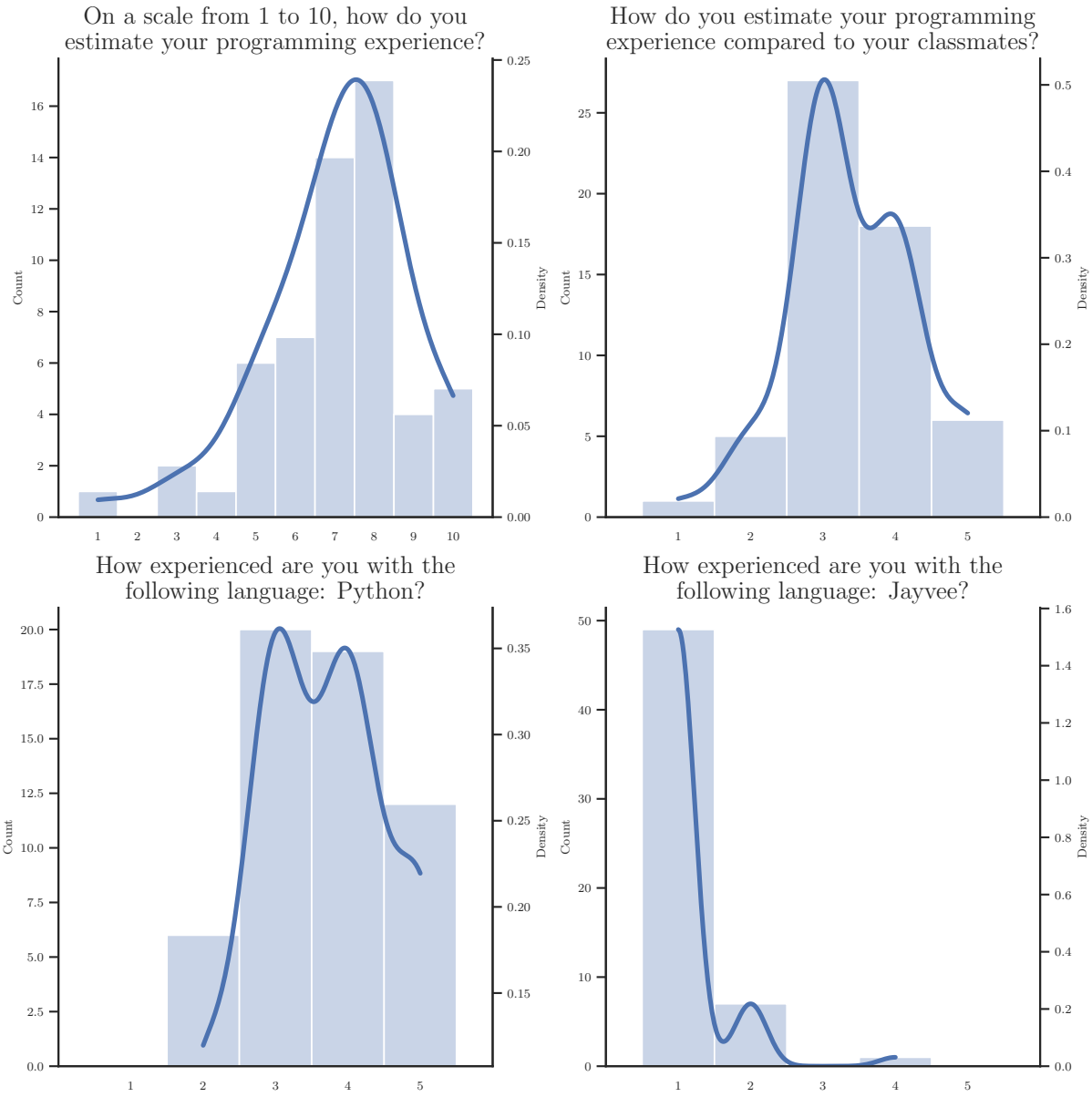


Figure 8: Previous experience of experiment participants.

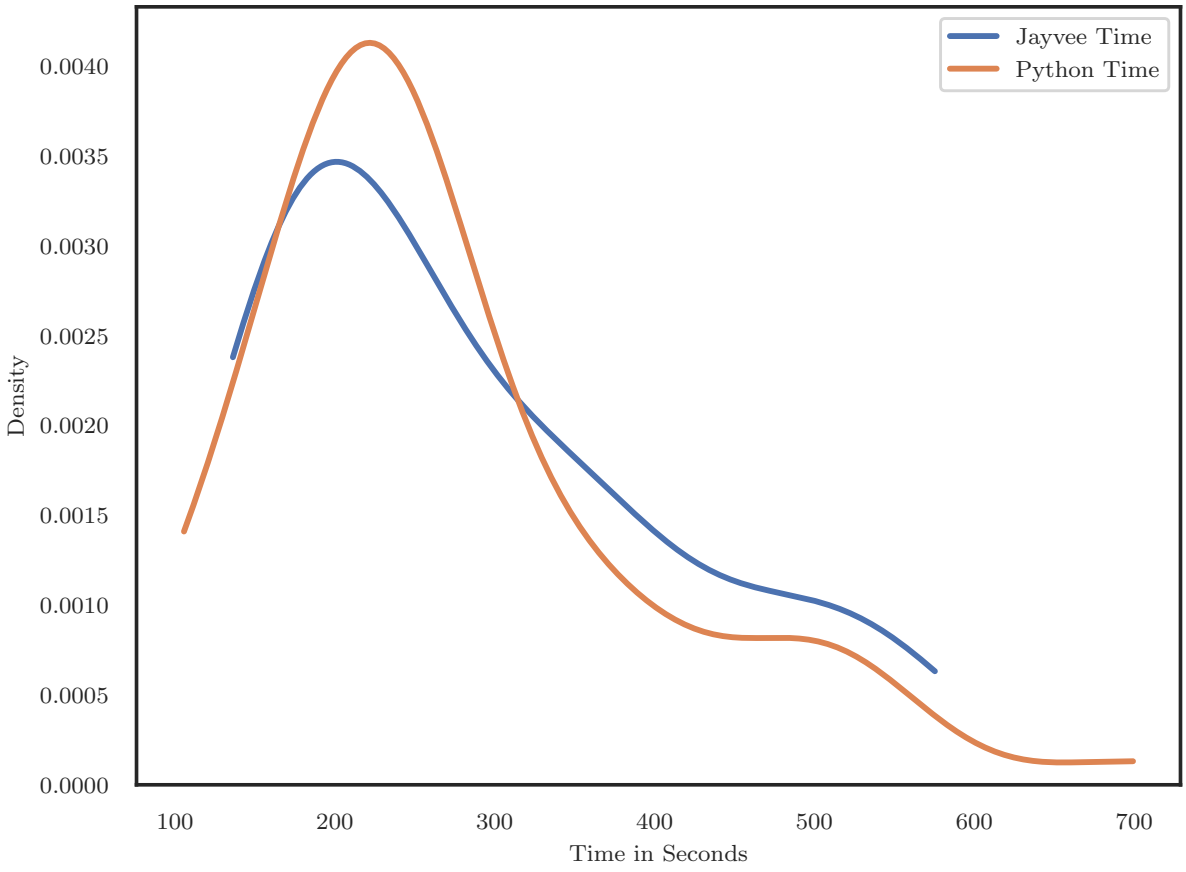


Figure 9: Kernel-density-plot of time on task for Jayvee compared to Python/Pandas.

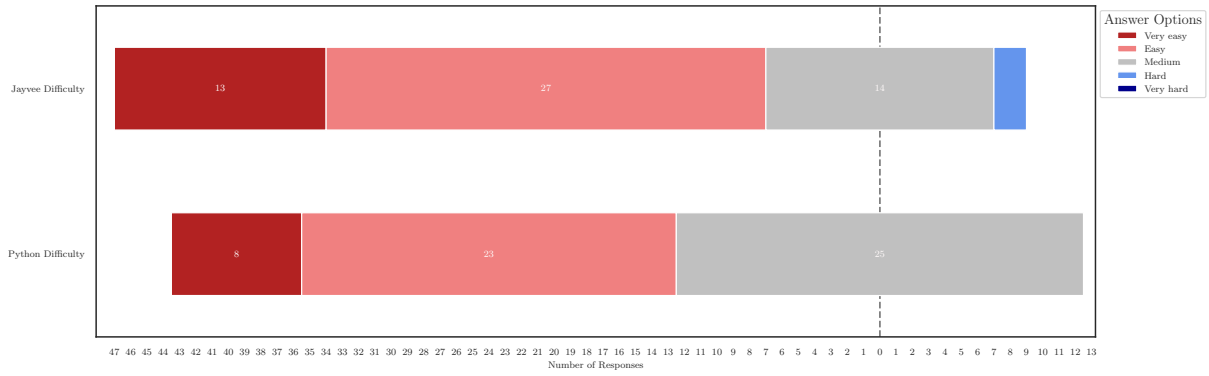


Figure 10: Diverging stacked bar charts according to Robbins et al. (2011) and Heiberger and Robbins (2014) for perceived difficulty of using Jayvee compared to Python/Pandas.*

* One outlier participant (S25) considered using Jayvee hard (and Python/Pandas easy) due to their lack of previous experience with Jayvee and did not provide more details, writing: "(Jayvee) is new so I think it was not easy to understand or read."

E

Paper 5: Is spreadsheet syntax better than numeric indexing for cell selection?

In this appendix, the paper *Is spreadsheet syntax better than numeric indexing for cell selection?* is reproduced in full. The manuscript is currently submitted and undergoing peer review. A preprint is published as:

Heltweg, P., Riehle, D., & Schwarz, G.-D. (2025). Is spreadsheet syntax better than numeric indexing for cell selection? In arXiv [cs.PL]. arXiv. http://arxiv.org/abs/2505.23296 [18]

[1] *https://creativecommons.org/licenses/by/4.0/*

IS SPREADSHEET SYNTAX BETTER THAN NUMERIC INDEXING FOR CELL SELECTION?

PREPRINT

Philip Heltweg
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
philip@heltweg.org

Dirk Riehle
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
dirk@riehle.org

Georg-Daniel Schwarz
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
georg.schwarz@fau.de

May 30, 2025

ABSTRACT

Selecting a subset of cells is a common task in data engineering, for example, to remove errors or select only specific parts of a table. Multiple approaches to express this selection exist. One option is numeric indexing, commonly found in general programming languages, where a tuple of numbers identifies the cell. Alternatively, the separate dimensions can be referred to using different enumeration schemes like "A1" for the first cell, commonly found in software such as spreadsheet systems.

In a large-scale controlled experiment with student participants as proxy for data practitioners, we compare the two options with respect to speed and correctness of reading and writing code.

The results show that, when reading code, participants make less mistakes using spreadsheet-style syntax. Additionally, when writing code, they make fewer mistakes and are faster when using spreadsheet syntax compared to numeric syntax.

From this, a domain-specific syntax, such as spreadsheet syntax for data engineering, appears to be a promising alternative to explore in future tools to support practitioners without a software engineering background.

Keywords domain-specific languages, data engineering, programming syntax, controlled experiment, empirical study

1 Introduction

A common challenge in data engineering is working with unstructured, two-dimensional data as it can be found in CSV files or spreadsheet software. Especially data sets based on exports from spreadsheets made for human readers have to be wrangled without the easy-to-use format that would allow for the selection of cells by column names or other structured tools.

In these cases, data is organized for human readers to consume rather than machines. Often, values are distributed in a 2D data structure to place them in a 2D space when displayed as part of a sheet. While these data sets are technically machine-readable, they have to undergo extensive data engineering work before they are available in a format that is easily importable into, e.g., a Pandas dataframe.

General-purpose programming languages (GPLs) can be used to manipulate this data, e.g., to select a subsection of it or remove errors. However, the syntax used in most GPLs has its origin in the numeric and often zero-indexed access of array structures. To select cells using this syntax, programmers refer to columns and rows by numbers or

numeric ranges with the distinction between both being mostly by position. For example, Pandas/Python cell selection is performed by the axis of the underlying dataframe so that `df.iloc[0, 1]` selects the cell in the first row (on index 0) and second column (on index 1).

While this syntax is familiar to professional software engineers, practitioners from adjacent fields that also work with two-dimensional (2D) data often use spreadsheet software to manipulate cells visually. Popular tools such as Microsoft Excel or Google Sheets show numbers as the reference to rows and characters to refer to columns, thereby syntactically separating the indexing dimensions. Therefore, the equivalent reference string to the Python/Pandas example above would be `B1`. In this syntax, columns are expressed as characters (`B` for the second column) while references to rows stay numeric, but their index starts more naturally at one instead of zero.

With these different syntactical approaches, data practitioners with a background in using spreadsheet software to manipulate and clean their datasets can struggle when reading or writing code using numeric indexing. Mistakenly switching the order of row and column references or forgetting about zero-indexing can lead to crashes or subtle errors in the resulting datasets.

As an alternative, domain-specific languages (DSLs) have been shown to be a potential middle ground between GPLs and visual tools, enabling domain experts to efficiently contribute in a wide range of domains outside of data engineering [16, 10], as well as when building data pipelines [5]. They do so by re-using concepts and conventions from the domain they cover. It stands to reason that addressing cells in unstructured 2D data sets could be easier for data practitioners using spreadsheet syntax rather than numeric indexing.

In this study, we conduct a controlled experiment to test this hypothesis and provide a basis for further research. Based on quantitative data from a large group of student participants, we compare the use of spreadsheet-style syntax in a DSL for data engineering with the numeric syntax of Pandas, an industry-standard data engineering library for Python.

Our goal is to answer the following research questions:

Research Question 1: Does spreadsheet-style syntax have an effect on *bottom-up program comprehension of* cell selection in unstructured 2D data by data practitioners compared to numeric syntax...

a: regarding speed?

b: regarding correctness?

Research Question 2: Does spreadsheet-style syntax have an effect on *code creation* for cell selection in unstructured 2D data by data practitioners compared to numeric syntax...

a: regarding speed?

b: regarding correctness?

With this study, we contribute:

1. The results of hypothesis tests in a controlled experiment on the effects of using spreadsheet-style syntax instead of numeric syntax for cell selection, providing a foundation for future studies towards the ideal syntax for DSLs in data engineering.
2. A detailed description and accompanying code release of an experiment instrument to run controlled experiments for cell selection that enables reproduction and re-execution of the experiment with data professionals.

This article is organized as follows:

First, we provide a short overview of related studies in section 2, then we present the research design in section 3, followed by the results in section 4. We provide additional context and insights in the discussion in section 5. Limitations to the results are presented in section 6 before a summary of the insights and future work in section 7.

2 Related Work

Empirical studies of programming language design, such as different syntax, with users are generally rare, even though they can lead to a deeper understanding that can not otherwise be achieved [2].

The relatively low amount of controlled experiments is a challenge in the wider field of software engineering research as well [14, 23].

The benefits of using domain concepts with DSLs have been investigated in a series of controlled experiments and replication studies by Kosar et al. They compare DSLs with GPLs and domain-appropriate libraries in a variety of

domains from GUI programming to feature diagrams [15, 17, 16]. The domain-specific concepts allow participants to work more accurate and efficient, both with and without IDE support.

However, based on their experience they point out that the evaluation of DSLs and their features is domain-specific and should be done for each domain. To the best of our knowledge, our study is the first contribution to this effort for cell selection in data engineering.

Hoisl et al. [6] compare notation for scenario-based testing (two text-based and one diagrammatic) and find the more natural-language based one to outperform others, including a structured language. Their work is an indication that task outcomes can be improved by syntax that is more familiar to the participants that we test in a separate domain.

How deeply domain concepts should be part of a DSL was previously studied by Häser et al. [7] for behavior driven development who compared a simple DSL with a DSL that was enriched with domain concepts. They find that participants can complete tasks significantly quicker with the DSL that includes domain concepts without an effect on quality.

Recently, Klanten et al. [13] have evaluated a similar approach to using domain-specific syntax (a domain-specific syntax for type inference rules compared to an implementation in Java) with positive effects for speed and correctness.

Similar to these studies, our experiments contributes additional data towards the study of optimal DSL features in a specific domain, in this case with regards to cell selection syntax in data engineering.

3 Research Design

We gathered quantitative data using a controlled experiment with human participants. To do so, we first defined a plan for the experiment that we refined iteratively with pilot experiments with other researchers. Then, we executed the experiment in-person during multiple sessions on one day. Finally, we analyzed the data using standard statistical methods.

The following structure is adapted from the proposed guidelines for reporting controlled experiments as suggested in Wohlin et al. [25], adapted from Jedlitschka and Pfahl [9].

3.1 Problem Statement

Working with otherwise unstructured 2D data is a common task in data engineering, for example when handling data from CSV files. When the data is formatted like a table with a clear header row, column names can be used to select subsets. However, often header data is missing or complex, multi-line headers based on exports from human-readable sheets make simple indexing by column names impossible. In those cases, subsets of data need to be selected first to be extracted, deleted, or transformed in follow-up steps. Different ways to manipulate these data structures exist:

|

1. General-purpose programming languages with libraries, such as Python and Pandas, using numeric indexing like iloc, e.g., `df.iloc[1:5, 2:4]`
2. Domain-specific spreadsheet software like Microsoft Excel or Google Docs and their syntax, e.g., `A1:B10`

While the numeric indexing might be familiar to software developers and especially users of Python/Pandas, spreadsheet-software is widely used by subject-matter experts and data practitioners in data engineering. For a domain-specific language for data engineering that aims to enable subject-matter experts to contribute, it is unclear whether using numeric indexing or spreadsheet-software formula syntax is the better choice to select subsets of 2D data structures.

3.2 Research Objectives

We follow the Goal/Question/Metric template [25, 1] to define the research objective:

- Analyze *two cell selection syntaxes*
- for the purpose of their effect on *bottom-up program comprehension and code creation for cell selection*
- with respect to *speed and correctness*
- from the point of *view of researchers*

- in the context of *a university course with masters level students learning data science (as proxies for data practitioners)*

3.3 Context

The context of the experiment is a master's level course teaching advanced methods of data engineering. Students were largely from master's degrees in artificial intelligence and data science. Controlled experiments with students allow for the initial evaluation of hypotheses that can be extended by experiments with practitioners in future research if they are a conscious choice as a representative of a population [3, 20].

We consider this cohort of students as an appropriate proxy for our target population of data practitioners that are used to working with data as part of their job but do not have a professional programming background. Similar to them, students from artificial intelligence and data science have heard lectures on statistics and theoretical algorithms but lack experience in professional software development. From previous work with a similar cohort of students, we know that they also use spreadsheet software to work with data in addition to writing scripts [5].

The participants learned an open-source domain-specific language called Jayvee [5] using spreadsheet-style cell selection syntax during the course. They did so by listening to introductory lectures and completing data engineering exercises while implementing a self-directed data science project in Python.

3.4 Experimental Design

3.4.1 Goals, Hypotheses, Parameters, and Variables

We derived two goals from the research objectives.

Goal 1: Understand if the use of spreadsheet-software cell selection syntax has an effect on bottom-up program comprehension (pc) compared to numeric indexing in regard to

a. speed

b. correctness

Goal 2: Understand if the use of spreadsheet-software cell selection syntax has an effect on code creation (cc) compared to numeric indexing in regard to

a. speed

b. correctness

During the experiment, we defined variables and controlled the following parameters.

Parameters

1. The tasks, based on two real open data sets. We selected two different, real data sets that were understandable without any special domain knowledge and slightly adapted them by removing rows with empty values, selecting a subset of 10 by 10 cells and randomizing the order of rows.
2. The students, from a master's level university course on data engineering.
3. The programming environment, an in-person experiment on a web-based experiment tool. The experiment tool did not provide syntax highlighting or auto completion of any sort and ensured that individual editor choice had no influence on the results.
4. Available help, only allowing standard documentation. We asked experiment participants to open the official documentation of both treatments before the start of the experiment and to not use the internet in any other way. One researcher ensured that participants did not leave the provided experiment environment at all times.

Independent variables

1. The cell selection syntax used, either a DSL with *Spreadsheet* syntax or Python with Pandas and it's iloc cell selection using *Numeric* syntax

Dependent variables

1. The average *time* to task completion, in seconds from the moment a participant started a task until they decided to submit their solution.
2. The average correctness of solution defined as defined by the Jaccard index [8]: $J(C,S) = \frac{|C \cap S|}{|C \cup S|}$ where C is the set of cells that should be selected and S the set of cells that are selected.

From the goals and based on the variables we measure, we defined the following hypotheses to test.

For goal 1a, the null hypothesis is *"Using spreadsheet-software cell selection syntax has no effect on the speed of bottom-up program comprehension compared to numeric indexing"*, more formally:

$$\begin{aligned} H_{0,1a} &: time_{pc}(Spreadsheet) = time_{pc}(Numeric) \\ H_{1,1a} &: time_{pc}(Spreadsheet) \neq time_{pc}(Numeric) \end{aligned} \tag{1}$$

For goal 1b, the null hypothesis is *"Using spreadsheet-software cell selection syntax has no effect on the correctness of bottom-up program comprehension compared to numeric indexing"*, more formally:

$$\begin{aligned} H_{0,1b} &: correctness_{pc}(Spreadsheet) = correctness_{pc}(Numeric) \\ H_{1,1b} &: correctness_{pc}(Spreadsheet) \neq correctness_{pc}(Numeric) \end{aligned} \tag{2}$$

For goal 2a, the null hypothesis is *"Using spreadsheet-software cell selection syntax has no effect on the speed of code creation compared to numeric indexing"*, more formally:

$$\begin{aligned} H_{0,2a} &: time_{cc}(Spreadsheet) = time_{cc}(Numeric) \\ H_{1,2a} &: time_{cc}(Spreadsheet) \neq time_{cc}(Numeric) \end{aligned} \tag{3}$$

For goal 2b, the null hypothesis is *"Using spreadsheet-software cell selection syntax has no effect on the correctness of code creation compared to numeric indexing"*, more formally:

$$\begin{aligned} H_{0,2b} &: correctness_{cc}(Spreadsheet) = correctness_{cc}(Numeric) \\ H_{1,2b} &: correctness_{cc}(Spreadsheet) \neq correctness_{cc}(Numeric) \end{aligned} \tag{4}$$

We chose two-tailed hypotheses because we had no prior knowledge about the effect direction that we expected.

3.4.2 Experiment Design

We chose a crossover design for our experiment, a within-subjects design in which each participant is assigned to every treatment. We chose a crossover design because students can have different previous experiences which could lead to challenges when measuring differences between participants groups instead of differences to the participants average [23]. In addition, crossover designs are commonly used in software engineering research and well understood [26].

However, because each participant is assigned to all treatments, crossover designs can introduce carryover effects in which experience from previous tasks influences the completion of future tasks. To reduce this effect, we assigned participants to two different sequences and introduced an initial non-tracked task in pseudocode that allowed them to get familiar with the experiment tool instead of having to learn it during the first real tasks.

Participants were randomly assigned to two sequences AB (first spreadsheet syntax, then numeric indexing) and BA (first numeric, then spreadsheet). To study the effects on both program comprehension and code creation, we ran two different sets of tasks. A short description of the goal of each task is shown in Table 1. For both code creation and program understanding, our goal was to offer one task that includes full rows, one that includes full columns, and two tasks that handle different subsets of cells.

In one session, each participant completed both sets of tasks with the following task order: For code creation, task 1 to 4, see Table 2. For program comprehension, task 5 to 8, see Table 3.

To reduce learning effects between the sets of tasks, the dataset used for program comprehension was different from the one for code creation. In any case, the datasets were real-world open data sets, slightly edited to remove header information and empty values, randomize the order of columns, and standardize them to a 10 by 10 grid.

Participants were assigned randomly to a sequence using JavaScript's built-in `Math.random` method when opening the experiment tool.

Table 1: Task descriptions.

Task	Description
1	Understand code that selects complete rows
2	Understand code that selects connected cells, no complete rows/columns
3	Understand code that selects complete columns
4	Understand code that selects connected cells, no complete rows/columns
5	Write code to select complete rows
6	Write code to select connected cells, no complete rows/columns
7	Write code to select complete columns
8	Write code to select connected cells, no complete rows/columns

Table 2: Sequences and intervention assignment for code creation.

Sequence	Period			
	Task 1	Task 2	Task 3	Task 4
AB	Spreadsheet	Numeric	Spreadsheet	Numeric
BA	Numeric	Spreadsheet	Numeric	Spreadsheet

3.5 Participants

Participants for the experiment were selected by convenience sampling from students of a master's level course for advanced methods of data engineering, mostly from master's degrees in artificial intelligence and data science with some students from related degrees such as information systems. Students were familiar with both treatment syntaxes by previous participation the course.

Approval by an ethics committee is not required by our institution and not standard for these kinds of studies as they do not carry personal risk or undue burden. However, we shared an informed consent handout with participants detailing experiment goals, data handling and process by email. Immediately before the experiment started, we again shared the same informed consent handout, allowed time for questions and asked for an explicit opt-in while making it clear that not participating at this point would have no negative consequences.

The handout made explicit, that the students' performance in the experiment tasks had no effect on their course grade, however, we incentivized participation by rewarding points for the course grade for participation, independent of their performance or whether they opted into the use of their data for research purposes. Opting in to allow the use of their data was purely voluntary and had no effect on the grades of students.

We asked for permission to use the data before starting the experiment to make the opt-in independent of the performance during the tasks.

3.6 Objects

The experiment was carried out using a web-based experiment tool with a small CSV data set displayed as a table without header.

There were two types of tasks: code creation and code understanding. Before every task, a not-tracked example task in pseudocode allowed the participants to learn how the experiment tool worked and what would be expected of them. Additionally, the experimenters demonstrated the different task types at the start of the experiment.

First, for code creation, participants were shown the data on the left side of the tool. For every tasks, a different subset of the data was highlighted in blue. On the right side, participants were shown a short program excerpt (a snippet from the experiment tasks in the DSL is shown in Listing 1, numeric syntax tasks used equivalent Python/Pandas code) and asked to complete a code block selecting the highlighted cells, either using numeric indexing with the iloc API in Python/Pandas or spreadsheet-software syntax in the DSL.

Table 3: Sequences and intervention assignment for program comprehension.

Sequence	Period			
	Task 5	Task 6	Task 7	Task 8
AB	Spreadsheet	Numeric	Spreadsheet	Numeric
BA	Numeric	Spreadsheet	Numeric	Spreadsheet

Listing 1: A DSL code snippet participants had to complete for a code creation task. An input field after the range keyword allows for spreadsheet-style syntax to select cells.

```
// Other blocks and pipeline definition...

block DataSelector oftype CellRangeSelector {
  select: range                    ;
}
```

For numeric syntax using Pandas/Python, code creation tasks where based on the participants selecting cells based on position using the `iloc` API. The surrounding Python/Pandas code was provided during the experiment tasks so that participants only needed to use the numeric syntax inside the `iloc` call to complete the selection.

Using `iloc`, participants can select a subset of a Pandas dataframe using a variety of ways such as integers, arrays of integers or slice objects to refer to cell positions. During the lectures in preparation for the experiment the use of mainly integers or slice objects was highlighted (for example `df.iloc[0, 1]` or `df.iloc[1:3]`).

The DSL used in the experiment is based on connecting small blocks of computation using pipes. These blocks have to be configured by the user, with a block named `CellRangeSelector` allowing the selection of a subset of cells from 2D data. The syntax used to select a range of cells using this block aligns with common spreadsheet programs where ranges are described from the starting cell to a final cell (for example, `A1:B2` refers to the range from cell `A1` to `B2`).

Cells are referred to either by a character for the column, followed by a one-indexed number for the row (for example `B2` for the second column and second row). Additionally, either the column or the row reference can be replaced by a `*` to indicate the last cell in that row or column, allowing a syntax like `A1:B*` to select all cells in the first two columns of a data set.

In the same way as for the numeric syntax, all custom code for the DSL was provided and did not have to be remembered by the participants. They only needed to complete the `select` property of a block using cell selection syntax.

An example screenshot of the whole task screen in the experiment tool is shown in Figure 1.

The second type of task was aimed at testing bottom-up program comprehension. Participants were shown code that selects a subset of cells from a 2D data structure in either the DSL or Python/Pandas (using numeric indexing, an example code snippet in Python from the experiment is shown in Listing 2, equivalent DSL code was shown for spreadsheet syntax).

On the right, participants were shown the actual data in the same view as during the code creation tasks. They could highlight cells in blue by either clicking them or dragging the mouse and were asked to highlight the cells that will be selected with the code shown.

Listing 2: Python code excerpt for a program comprehension task

```
# Python code
# Imports and pipeline definition...

df = pd.read_csv('./data.csv')

df.iloc[6:10, 0:3]
```

A complete task view for program comprehension is shown in Figure 2.

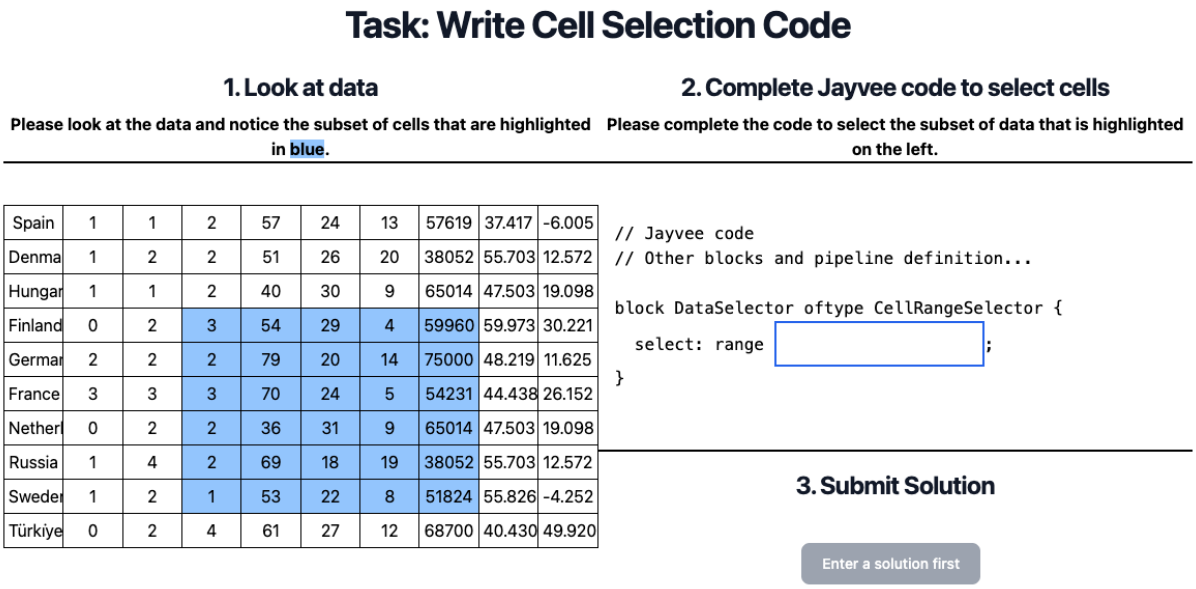


Figure 1: Code creation task in the experiment tool.

3.7 Instrumentation

Before the experiment, participants were trained in Python and the DSL and their respective cell selection strategies during the university course.

Participants were introduced to the DSL and spreadsheets software cell selection syntax in two introduction lectures. During the semester, they completed five exercises with real-world open data using the DSL.

Based on their backgrounds, some participants already had some prior knowledge in Python and Pandas. In addition, they implemented a self-directed, real-world data science project in Python. The project included writing an automated data pipeline to download, clean and save open data sets before using the data to create a report.

I preparation for the experiment, we held a lecture on how to positionally select cells from 2D. The lecture included information about cell selection both in Python/Pandas using the iloc API used in the tasks as well as in the DSL using the same block used in the tasks.

The measurement instrument was a browser based environment that automatically tracked events for future analysis. Participants were asked not to use other programs or leave the experiment environment in any way. This explicitly included using websites to get outside help such as search engines or AI services. One experimenter monitored the screens of participants to make sure that they followed the directions.

At the start of the experiment, participants were allowed to open links to the official documentation of position-based indexing by Pandas, as well as the cell range documentation by the DSL. These links were provided by the experiment tool and the same for all participants.

3.8 Data Collection Procedure

The experiment was conducted in person in computer labs with identical equipment provided by the university. Due to the large size of the experiment, multiple sessions were conducted immediately following each other on one day and students were asked not to share the experiment setup with later groups.

After a brief introduction, the informed consent letter was handed out. The same letter was already shared by email beforehand. Participants were given the chance to ask questions or withdraw from the experiment without any negative consequences.

Then, one researcher followed a predefined experiment procedure document to present the experiment tool, the experiment flow, and two example tasks with pseudocode. Participants were asked to focus on correctness over speed if in doubt.

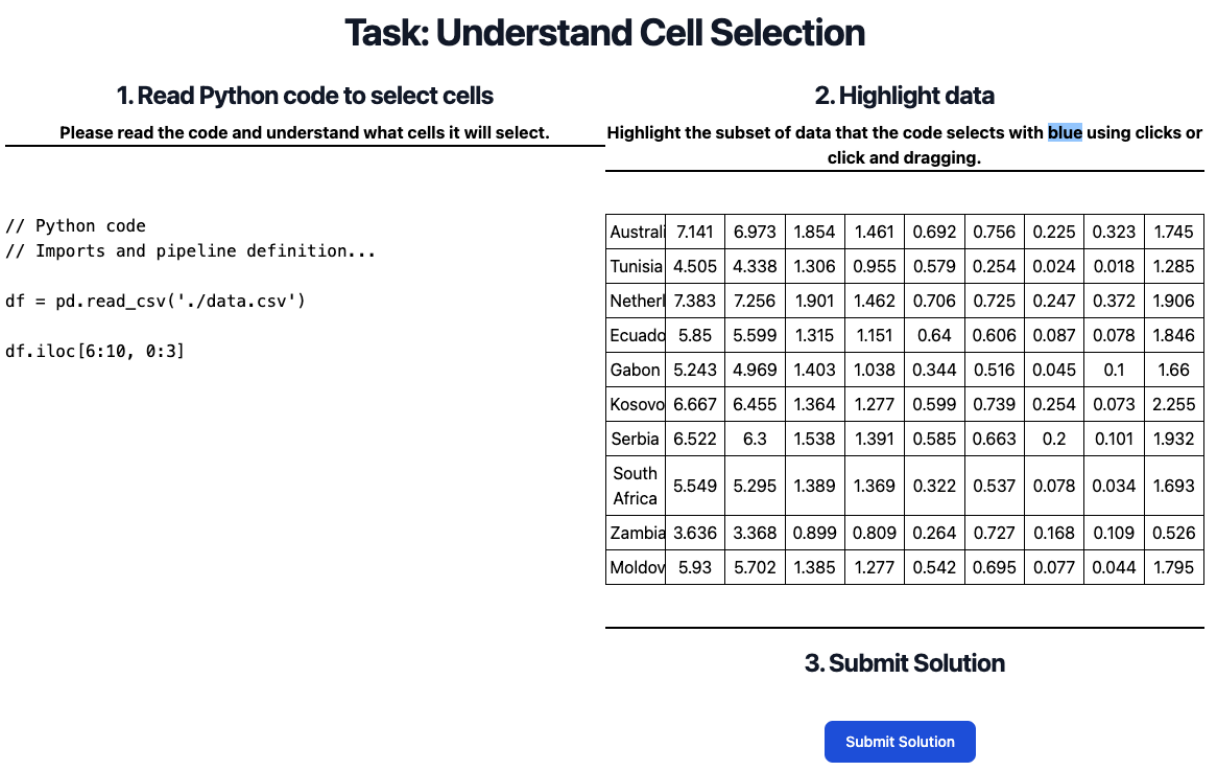


Figure 2: Program comprehension task in the experiment tool.

Data collection was done automatically using the experiment tool. For this, the tool automatically recorded events, their timing, and associated data like the submitted solution.

For each task, participants decided themselves when they considered a solution complete and submitted it. Between tasks, a success screen allowed them to pause before attempting the next task.

Participants were given 40 minutes to complete all tasks, with an announcement of time passing every 10 minutes.

3.9 Analysis Procedure

The analysis was completed using Python 3.11 with Pingouin 0.5.5 [21].

First, experiment data was anonymized, and data integrity checks were performed, verifying that the tracked events are in the expected order and quantity. Another round of data integrity checks (such as verifying that no correctness value was outside the bound of 0 to 1) was executed after calculating the derived variables.

Second, from the timestamps of task start and end, the duration of a task was calculated in seconds. The overall time for a treatment is then taken as the average of both tasks completed using that treatment.

Correctness was calculated automatically by executing the code written by the participant and comparing the selected cells with the correct ones by building the Jaccard index, then averaging the correctness of both tasks. The more rigid structure of the DSL leads to slightly lower correctness values for code creation in the spreadsheet syntax for rare edge cases. We took note of this; however, with the correctness of the spreadsheet syntax being higher on average, this effect ultimately had no influence on the results of the hypothesis tests.

Few code creation submissions included syntax elements that were already part of the program snippet that was shown to participants, such as a trailing semicolon for the DSL or superfluous brackets for Pandas/Python. To reduce the amount of basic syntax errors that lead to correctness values of 0 with otherwise correct solutions, these syntax elements from the experiment tool were automatically removed for both Python and the DSL.

For the derived variables, outliers were marked using standard 1.5 interquartile range (IQR). Experiment runs with outliers in the variables under consideration were removed for the hypothesis test.

4 Results

The experiment was performed with 100 participants, of which seven were removed due to an abnormal amount or order of events (mostly due to participants navigating back to previous tasks already completed during the experiment), leading to a final count of 93 valid experiment runs. Of these, 52 had been randomly assigned to the sequence BA, 41 to the sequence AB.

For the results, we chose to only consider experiment runs with no outliers in the variable under consideration, meaning individual hypothesis tests will have slightly different sample sizes depending on how many outliers were removed. Including experiment runs with outliers in the analysis slightly affects the individual values but does not change the results of the hypothesis tests.

We chose kernel-density plots to visualize the distributions of the variables because they provide a good overview of the distribution and make it easy to see non-normality [12]. The plots are cut off at the extreme data points so that only existing data points are graphed.

The resulting distributions were analyzed for normality using the Shapiro-Wilk test [19]. Since all distributions were non-normal, we used the Wilcoxon signed-rank test for paired data for further hypothesis tests [24, 25]. We used the standard $\alpha = 0.05$ as a measure of statistical significance.

The detailed results of all hypothesis tests are shown in Table 4 and Table 5. We use common language effect size (CLES) as a more intuitive measure of effect size, first introduced by [18], but based on the generalization by [22], to discuss effect sizes.

The CLES describes the probability of a random value from one distribution to be larger than one from the other. Therefore, a value of 0.5 is expected for no effect and larger deviations from that value with larger effects. We interpret CLES based on the guidelines in [22] as either small, medium, or large (calculating $1 - CLES$ for values below 0.5). For completeness, we additionally include effect sizes as matched pairs rank-biserial correlation (RBC) [11].

4.1 Program Comprehension

The results of the hypotheses tests for program comprehension are shown in Table 4.

Table 4: Wilcoxon signed-rank test results for program comprehension.

	n	W-val	p-val	RBC	CLES
H_{1a}	84	1459.0	0.146597	-0.182633	0.464427
H_{1b}	83	1230.0	0.018937	0.29432	0.541951

We defined the null hypothesis for speed of program comprehension, $H_{0,1a}$, as *"Using spreadsheet-software cell selection syntax has no effect on the speed of bottom-up program comprehension compared to numeric indexing"*. With a p-value of $p \approx 0.147$ ($n = 84$), we have no reason to reject the null hypothesis and accept it as-is.

From the distribution shown in Figure 3, it seems while participants using numeric syntax show more varied task completion times, both treatments have a similar peak submission time. Any existing difference is not large enough to be statistically relevant.

Regarding $H_{0,1b}$, *"Using spreadsheet-software cell selection syntax has no effect on the correctness of bottom-up program comprehension compared to numeric indexing"*, we reject the null hypothesis ($p \approx 0.019$, $n = 83$) and instead adopt the alternative hypothesis.

From the distribution plotted in Figure 4, it is clear that participants understood cell selection significantly more correct when using spreadsheet syntax. This result is potentially very relevant to practice, as any data practitioner that wants to work with existing scripts first has to understand them and be confident that they are selecting the correct data.

However, with a CLES of ≈ 0.54, the size of the effect is very small and has to be considered with that in mind. Further experiments should be conducted to verify that this effect does in fact exists.

4.2 Code Creation

The results of the hypotheses tests for code creation are shown in Table 5.

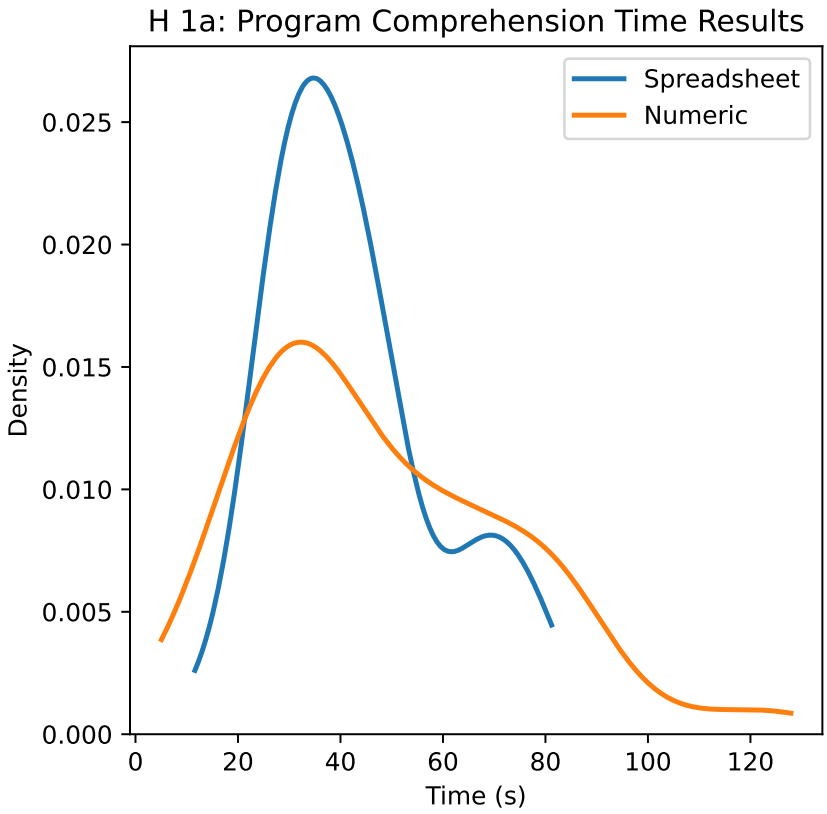


Figure 3: Kernel density plot for the results of H_{1a}, time, program comprehension

Table 5: Wilcoxon signed-rank test results for code creation.

	n	W-val	p-val	RBC	CLES
H_{2a}	84	163.0	4.775870e-13	-0.908683	0.160289
H_{2b}	93	463.5	0.000003	0.637324	0.658053

The null hypothesis $H_{0,2a}$, *"Using spreadsheet-software cell selection syntax has no effect on the speed of code creation compared to numeric indexing"*, shows the largest effect. Participants are significantly faster to write code using the spreadsheet syntax ($p \approx 4.776e-13$, $n = 84$, see also Figure 5).

In addition, the effect size of $1 - 0.16 \approx 0.84$ can be classified as large. Given how strong the effect is, using spreadsheet syntax to select cells from two-dimensional data has a clear impact on data practitioners and will allow them to complete their tasks faster.

Lastly, $H_{0,2b}$, *"Using spreadsheet-software cell selection syntax has no effect on the correctness of code creation compared to numeric indexing"* shows a statistically significant result as well ($p = 0.000003$, $n = 93$) with participants being more correct when using spreadsheet syntax to select cells instead of numeric indexing.

From the kernel-density plot shown in Figure 6 participants complete code creation tasks with much higher correctness using spreadsheet syntax than numeric syntax. Additionally, the rate of solutions with a correctness of 0 (mostly due to syntax errors) is dramatically lower when using the more simple spreadsheet syntax.

The effect size of ≈ 0.66 is medium, with a high relevance to practice. Working with two-dimensional data is a common task for data engineers, and increased correctness will lead to lower amounts of bugs and further reduce the time needed to arrive at a correct solution.

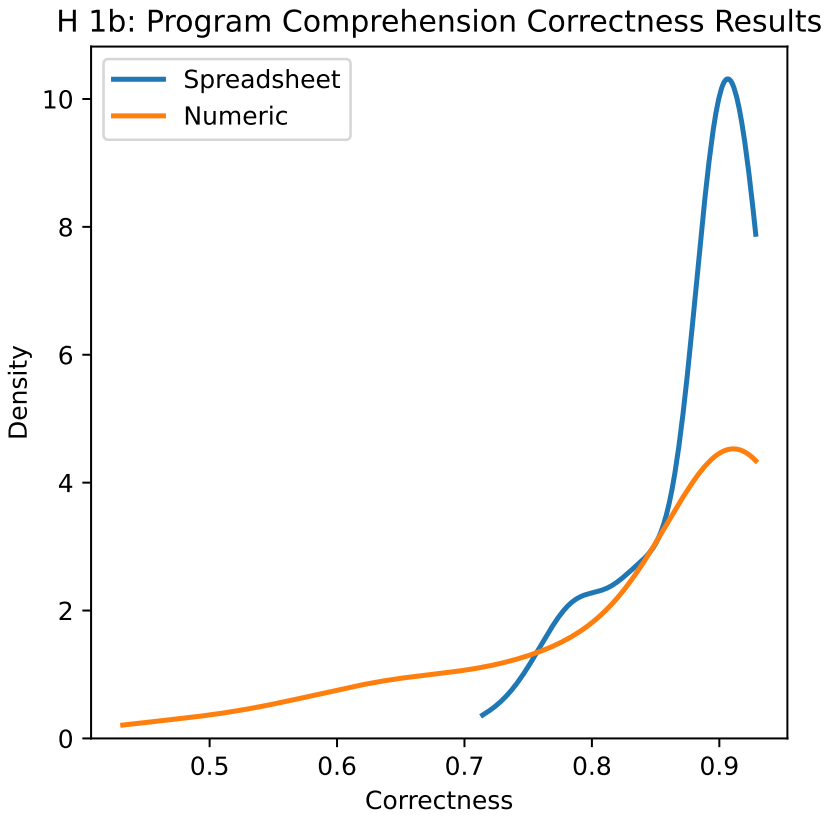


Figure 4: Kernel density plot for the results of H_{1b}, correctness, program comprehension

5 Discussion

In this chapter, we move beyond the quantitative results and discuss potential reasons for the effects that were measured in the experiment. Our goal is to provide additional insight from our extended engagement with the topic, the results and the participants; however, additional qualitative research should be done to rigorously describe causal explanations for the observed effects.

5.1 On Program Comprehension

With regards to program comprehension, using spreadsheet syntax resulted in higher correctness but had no statistically significant effect on the time needed.

Because the time needed to understand the cell selection syntax is similar, we assume participants largely did not try to verify their submission in depth by re-reading documentation and instead tried to answer from their intuitive understanding. It seems with this approach, both the spreadsheet style syntax as well as the numeric syntax are easy to read and did not create difficulties for the participants.

For correctness, the previous experience of participants very likely had an influence. During the coursework, many students pointed out that they often work with data sets using spreadsheet software like Microsoft Excel or Google Sheets. For many, these are standard tools used in data engineering to look at and edit small-scale 2D data sets. Similarly, practitioners from industry without a programming background often use spreadsheet software to work with data. Being able to reuse this previous experience can enable data practitioners to intuitively understand a selection syntax more correctly.

It also seems that choosing to start counting rows with one instead of sticking to the zero-indexing often found in GPLs means less confusion for users who do not have a programming background.

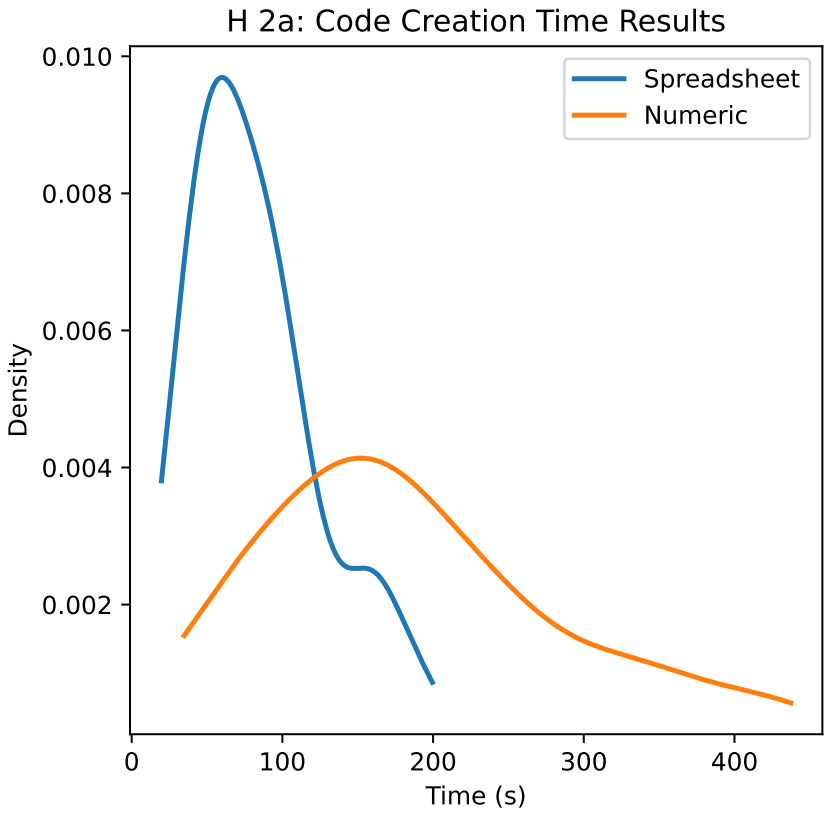


Figure 5: Kernel density plot for the results of H_{2a}, time, code creation

5.2 On Code Creation

When participants had to write code, they could complete the tasks in less time and with higher correctness when using the spreadsheet-style syntax compared to the numeric syntax.

A factor in the lower time needed to submit a solution might be the difference in the larger size of the documentation that is available for Pandas/Python. We encouraged the participants to, if in doubt, prioritize correctness over speed so it is conceivable that they verified their solution by re-reading the documentation.

On the other hand, this would also indicate that the spreadsheet syntax was more intuitive to understand and did require less double-checking with documentation.

It is important to point out that faster task completion is very likely at least partially predicated on the small scale of the dataset used. For larger data sets, especially when the spreadsheet syntax has to be extended and use two or more characters to refer to columns (e.g., the use of "AA" for the 27th column), the numeric syntax might be faster to use again.

Correctness was influenced by the comparatively larger amount of totally incorrect submissions with numeric indexing, which also includes syntax errors. The spreadsheet-style syntax has only one comparatively simple way to express a cell range. In contrast, the numeric syntax allows for some flexibility and is only one of many ways to select cells in Pandas. For example, some participants tried to refer to column names or used extraneous brackets or other not-allowed symbols, leading to syntax errors.

Aside from syntax errors, participants regularly made off-by-one errors using the numeric syntax, either incorrectly selecting more initial cells or missing a final row or column of cells. These errors seem to stem from a wrong intuition about zero-indexing selections in GPLs, as well as the resulting confusion about whether the final index is included or excluded from the selection. In contrast, the spreadsheet-style syntax seems to be more clear. One possible explanation is that the participants have previously selected data in the spreadsheet and visually seen the limits of their selection represented by the software.

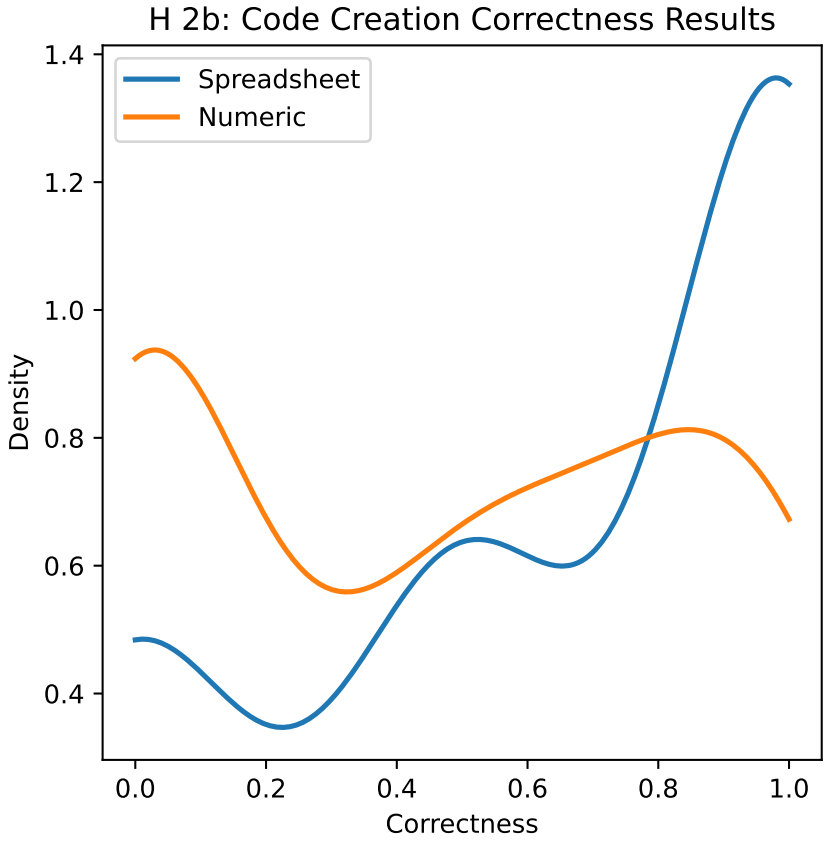


Figure 6: Kernel density plot for the results of H_{2b}, correctness, code creation

5.3 Potential Reasons for Improved Correctness

A potential factor for improved correctness, both when reading code and when writing it, is the fact that spreadsheet syntax uses two different numbering schemes to refer to the two distinct indices in two-dimensional data.

Going back to the initial example, numeric syntax such as accessing the cell in the first row and second column with Pandas/Python using `df.iloc[0, 1]` relies on the ordering of parameters. With this style, no additional context can be read from the syntax, and users have to rely purely on their knowledge about the correct position for, e.g., the index of the row they are trying to access.

In contrast, spreadsheet syntax such as the equivalent `B1` refer to columns using characters and rows by numbers. This way, users can directly read which part of the syntax refers to which concept without having to rely on previous knowledge about the implementation. Instead, they have to be aware of the convention of referring to columns by characters, which they are due to their background using spreadsheet applications.

The different syntaxes are grounded in the two competing mental models when thinking about multi-dimensional data. The programmer's view (that finds its expression in the numeric syntax used by GPLs) is based on multi-dimensional arrays or matrices that are accessed along their axis. This mental model scales better to more than two dimensions as it does not assign inherent meaning to any axis.

The alternative mental model for thinking about two-dimensional data that is used by many data practitioners is viewing the data in a spreadsheet. While this view does not scale past two dimensions, it allows for the assignment of meaning to each axis and, therefore, custom representations for each. Additionally, using spreadsheet software with the permanent visual representation of row and column labels as numbers and characters reinforces an intuitive understanding of them among practitioners.

5.4 Practical Implications

Overall, these results are an indication that a domain-specific syntax for cell selection should be considered when designing future languages for data engineering by data practitioners without a professional programming background.

The ability to contribute code faster hints at a lower technical barrier for users that have no background in software engineering, one of the major challenges to including subject-matter experts in collaborative data engineering [4]. Together with the reduced rate of errors, both when creating code and when understanding cell selections, these syntax adaptions could enable contributors with diverse backgrounds.

For language developers, knowing the target users' previous experience is most likely an important consideration. In this case, the insight that data practitioners often use spreadsheet programs to work with 2D data can directly lead to a more domain-appropriate syntax.

6 Threats to Validity

We followed a thorough research process to conduct this study. However, some potential threats to validity remain which we discuss according to the framework of threats to validity, as proposed in [25].

6.1 Threats to Conclusion Validity

To make a valid conclusion, one must understand the correct relationships between the treatment and the results of an experiment.

Since the authors of this study are also creators oft he DSL used as a treatment in this study, searching for positive results might have introduced bias. To mitigate this risk, we defined the hypotheses and the research design before the data collection. Further, we used standard research designs and statistical tests and report effect sizes and results regardless whether the results were statistically significant or not.

We employed a crossover experiment design to avoid the challenge of heterogeneity of students; in this manner, we measure differences in comparison to participants' average and not between participant groups [23].

For the data collection, we strictly followed a previously designed experiment procedure document to reduce individual bias while guiding participants through the experiment. We automated large parts of the experiment in the form of an experiment tool that implements the treatment and took measurements without interaction by the researchers.

However, subconscious bias remains a potential threat to the validity of the conclusions. Therefore, we share the experiment tool for a thorough review and invite independent replication studies.

6.2 Threats to Internal Validity

To attribute observed effects solely to the treatment, it is crucial to control for any extraneous factors that might influence the outcome.

One such external factor is the quality of the tools and tasks of the experiment. To ensure adequate quality, we tested the tool and the tasks in multiple sandbox tests with other researchers before using it during the experiment. We adjusted the tasks and the tool based on their feedback as suggested by Ko et al [14].

The participation in the experiment was voluntary for the class of students. The results might be biased by this selection if students expect that positive responses in regard to the DSL under study would positively influence their class grades. To avoid this, we clearly communicated that the data was anonymized and emphasized that neither participation nor performance in the experiment influenced their grade.

Another external factor to discuss is carryover effects between treatments. We addressed such potential learning effects by randomly assigning participants to different treatment orders and adding an initial task using pseudocode to allow familiarization with the tool and the task setup. Regardless of these measurements, we must recognize that carryover could still be an influencing factor on the results and aim for future replication with between-subject designs.

6.3 Threats to Construct Validity

To confirm that the measured variables accurately represent the intended theoretical constructs, it is essential to examine whether the operational definitions and instruments truly capture the underlying phenomena.

We clearly defined the dependent variables of the experiment and measured them programmatically. Further, we chose common measures for code comprehension and creation experiments: time and correctness [26]. However, while the Jaccard index we chose for correctness is a standard measurement, many competing definitions of correctness are possible.

Mono-method bias is a limitation for this study because we did only measure one variable for each construct. This presents a danger to insufficiently capture complex relationships, for example regarding program understanding. To strengthen the rigor of the results, additional experiments with more measurements would be needed. In future work, we plan to extend the current insights with more qualitative studies as well.

6.4 Threats to External Validity

To generalize the results of the experiment beyond its specific context, we need to carefully evaluate the applicability of the findings to other settings, populations, or times.

We chose students from master's degree programs in information systems, data science, and AI as participants for the experiment. For all drawn conclusions, it is important to contextualize them as representatives of a limited population, data practitioners that are not professional software engineers [3].

However, we believe that those students are good proxies for a population of subject matter experts working with data in the industry and represent the variety of data practitioners.

Additional experiments with real subject-matter experts would be needed to validate whether students are a proxy, but we expect the results to generalize well in this limited domain. We expect that the results do not generalize well to professional programmers with different previous backgrounds and more experiences with programming languages.

7 Conclusions

In conclusion, we conducted a large-scale, controlled experiment with student participants to find out if a domain-specific spreadsheet-style syntax had any effect on how well data practitioners select cells from 2D data, compared to the numeric syntax found in Pandas/Python.

In the experiment, participants completed tasks related to program comprehension by selecting a subset of cells as described by a program snippet. In addition, they had to complete a program with appropriate syntax to select the same cells that they were shown in a web-based tool.

With regards to program comprehension, we investigated if spreadsheet-style syntax had an effect on speed or correctness when reading cell selection code, compared to numeric indexing. Participants did understand the program more correctly when reading spreadsheet-style syntax but did not submit their solutions faster.

Similarly, for code creation, we measured time and correctness when completing program snippets with either spreadsheet-style syntax or numeric syntax. In these tasks, participants completed their tasks faster and more correctly when using spreadsheet-style syntax compared to numeric selection syntax.

From this data, we conclude that spreadsheet-style syntax can improve results for data practitioners when creating software artifacts for data engineering. Future language designers should consider the use of domain-specific syntax when targeting users who do not have a classical systems programmer background.

Concretely, the correctness of both reading and writing code was increased using spreadsheet syntax. This effect can improve the correctness of downstream data sets by reducing bugs in data pipeline code.

By using language syntax that is easier to use for practitioners and aligns more closely with their previous experience, technical barriers to participation by these users can be reduced. This in turn will allow more non-technical users, such as subject-matter experts, contribute to data engineering projects.

The implications for industry are important, with data engineering often consuming a large part of the costs for data science projects. Enabling contributors from a wider array of backgrounds to directly contribute with software artifacts can lower communication overhead and strain on professional software engineers.

While these results provide first quantitative indications, we can not draw clear causal explanations from them. To do so, additional qualitative research would be needed. By employing interviews or think-aloud protocols, the reasons for the effects of spreadsheet-style syntax should be explored in future work so that they can be used as guidelines for further language development.

Acknowledgments

This research has been partially funded by the German Federal Ministry of Education and Research (BMBF) through grant 01IS17045 Software Campus 2.0 (Friedrich-Alexander-Universität Erlangen-Nürnberg) as part of the Software Campus project 'JValue-OCDE-Case1'. Responsibility for the content of this publication lies with the authors.

Data Availability Statement

The data generated and analyzed during the current study is available on Zenodo at `https://doi.org/10.5281/zenodo.15543965`.

References

[1] V R Basili and H D Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[2] Raymond P L Buse, Caitlin Sadowski, and Westley Weimer. Benefits and barriers of user evaluation in software engineering research. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA, October 2011. ACM.

[3] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1):452–489, February 2018.

[4] Philip Heltweg and Dirk Riehle. A systematic analysis of problems in open collaborative data engineering. *Trans. Soc. Comput.*, 6(3-4):1–30, December 2023.

[5] Philip Heltweg, Georg-Daniel Schwarz, Riehle Dirk, and Quast Felix. An empirical study on the effects of jayvee, a domain-specific language for data engineering, on understanding data pipeline architectures. *Software: Practice & Experience*, 2025.

[6] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. Comparing three notations for defining scenario-based model tests: A controlled experiment. In *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 180–189. IEEE, September 2014.

[7] Florian Häser, Michael Felderer, and Ruth Breu. Is business domain language support beneficial for creating test case specifications: A controlled experiment. *Information and software technology*, 79:52–62, November 2016.

[8] P Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[9] A Jedlitschka and D Pfahl. Reporting guidelines for controlled experiments in software engineering. In *2005 International Symposium on Empirical Software Engineering, 2005.*, page 10 pp. Ieee, 2005.

[10] Arne N Johanson and Wilhelm Hasselbring. Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. *Empirical Software Engineering*, 22:2206–2236, 2017.

[11] Dave S Kerby. The simple difference formula: An approach to teaching nonparametric correlation. *Comprehensive psychology*, 3:11.IT.3.1, January 2014.

[12] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630, April 2017.

[13] Kai Klanten, Stefan Hanenberg, Stefan Gries, and Volker Gruhn. Readability of domain-specific languages: A controlled experiment comparing (declarative) inference rules with (imperative) java source code in programming language design. In *Proceedings of the 19th International Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, 2024.

[14] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, February 2015.

[15] T Kosar, N Oliveira, M Mernik, M João, M Pereira, M Repinåek, D Cruz, and P Rangel Henriques. Comparing general-purpose domain specific languages: empirical study. *Computer Science Information Systems*, 2010.

[16] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*, 23(5):2734–2763, October 2018.

[17] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17:276–304, 2012.

[18] Kenneth O McGraw and S P Wong. A common language effect size statistic. *Psychological bulletin*, 111(2):361–365, March 1992.

[19] S S Shapiro and M B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[20] W F Tichy. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering*, 5(4):309–312, 2000.

[21] Raphael Vallat. Pingouin: statistics in python. *The Journal of Open Source Software*, 3(31):1026, November 2018.

[22] András Vargha and Harold D Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and wong. *Journal of educational and behavioral statistics: a quarterly publication sponsored by the American Educational Research Association and the American Statistical Association*, 25(2):101–132, June 2000.

[23] Sira Vegas, Cecilia Apa, and Natalia Juristo. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering*, 42(2):120–135, February 2016.

[24] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80, December 1945.

[25] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science + Business Media, 2012.

[26] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 40 years of designing code comprehension experiments: A systematic mapping study. *ACM Comput. Surv.*, October 2023.