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.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2025 The Author(s). *Software: Practice and Experience* published by John Wiley & Sons Ltd.

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

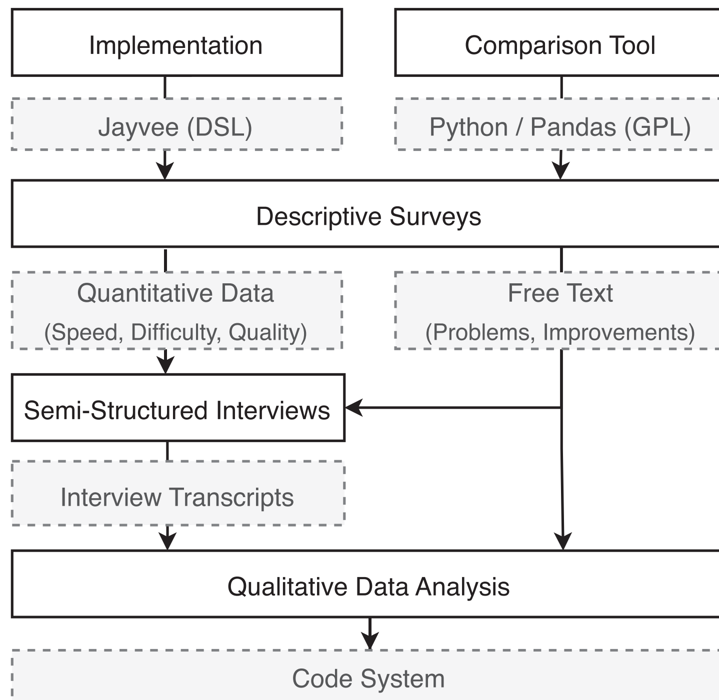

FIGURE 1 | Overview of the research design.

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

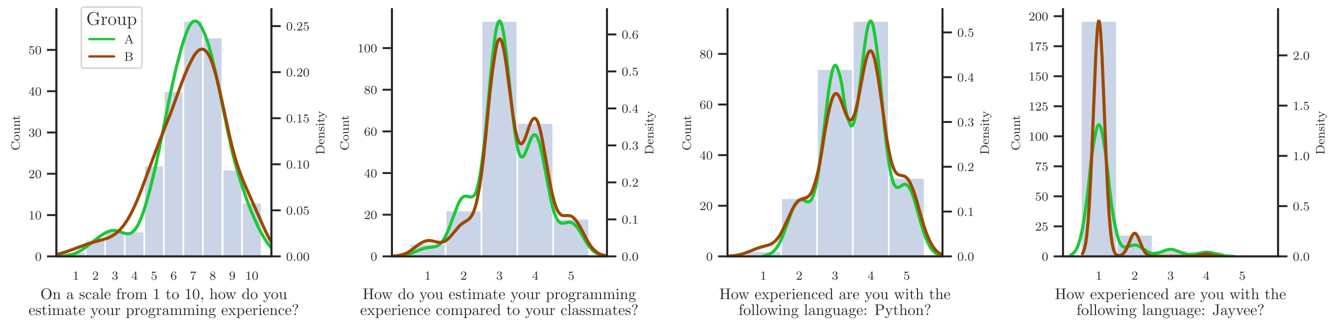

FIGURE 2 | Results regarding student's previous experience from the course entry survey.

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.

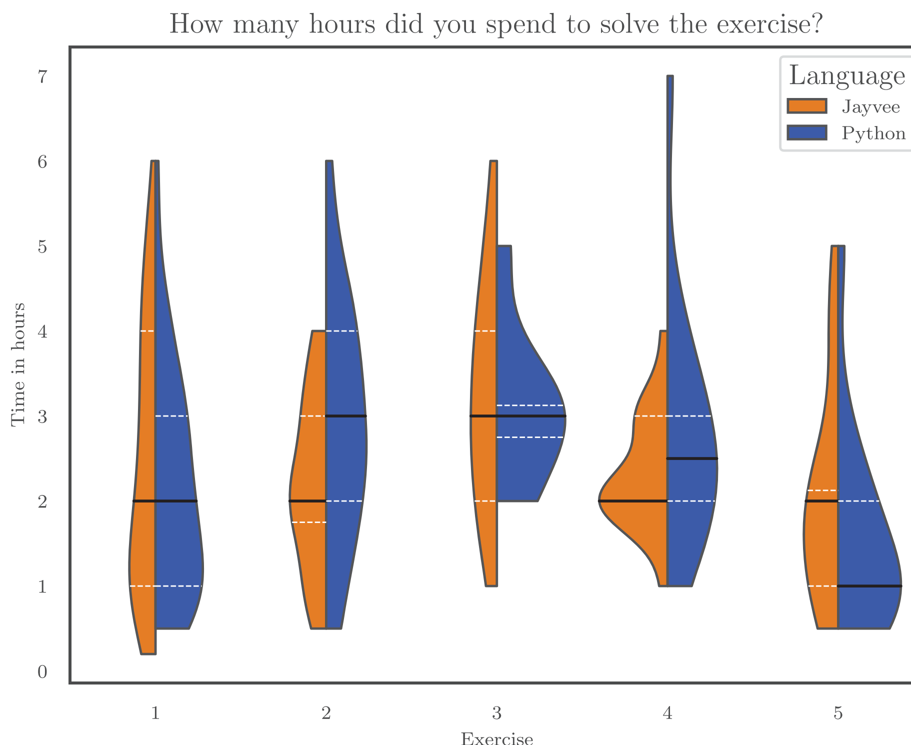

FIGURE 3 | Distribution of time spent per exercise, depending on the language used (lower is better). White bars represent Q_1 and Q_3, the black bar denotes Q_2.

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

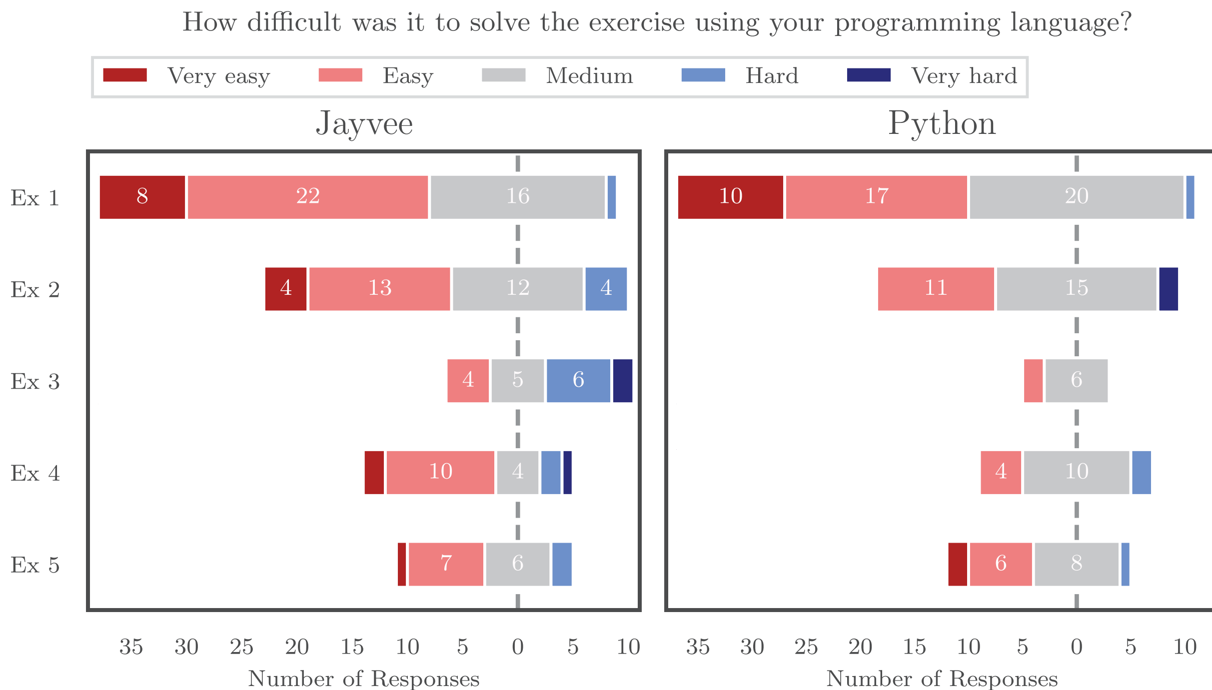

FIGURE 4 | Impressions of the difficulty of completing the exercise, depending on the language used (lower is better).

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

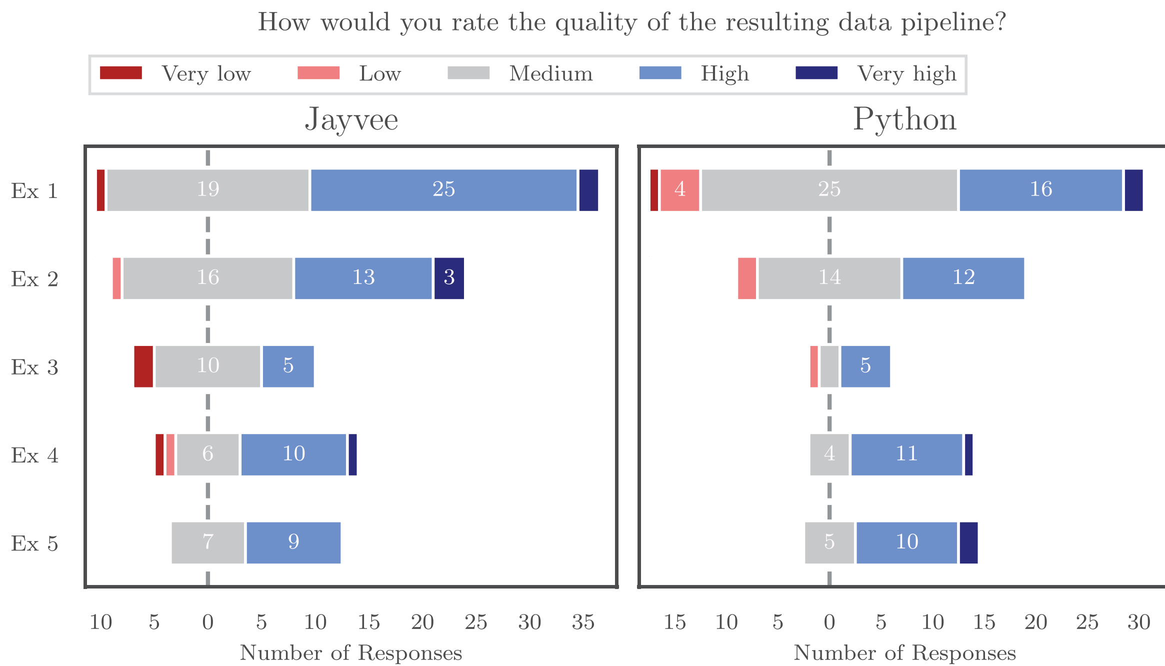

FIGURE 5 | Impressions of the quality of the resulting pipeline, depending on the language used (higher is better).

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

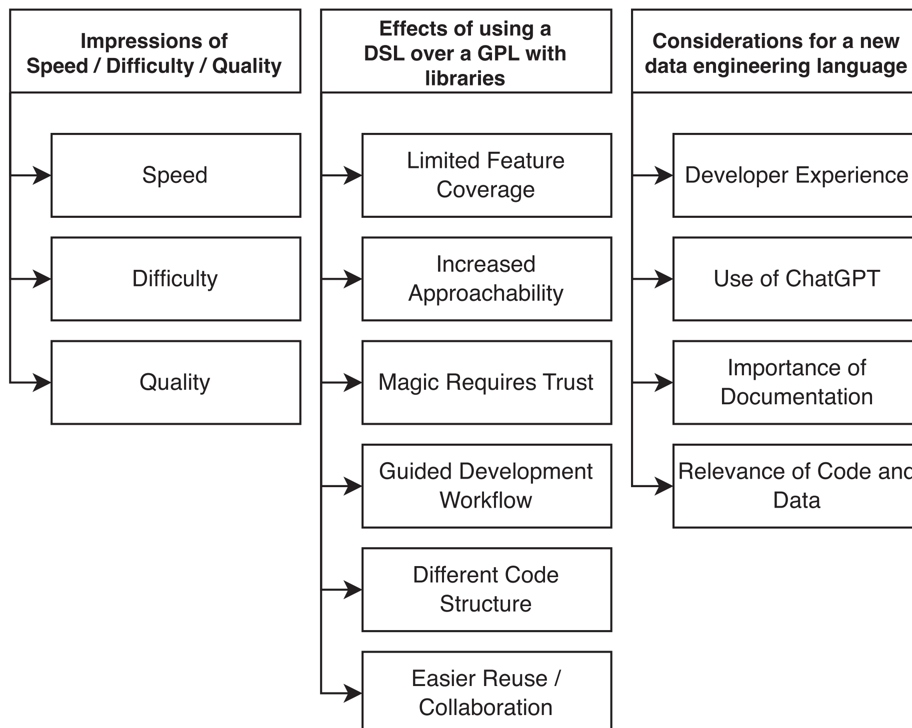

FIGURE 6 | Thematic map from thematic analysis of students' interviews and survey responses.

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.