Safe-DS: A Toolkit for Safe Development of Data Science Pipelines

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Lars Reimann
aus
Troisdorf, Deutschland

Bonn 2025

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

Gutachter/Betreuer:	Prof. Dr. Jens Lehmann
Gutachter:	Prof. Dr. Christian Bauckhage
Tag der Promotion:	2025-06-12
Erscheinungsjahr:	2025

Abstract

A typical development stack for data science (DS) programs, henceforth called *DS pipelines*, consists of Python as the language, various Python libraries, and a notebook environment where a pipeline is separated into cells that can be run independently. Notebooks may be edited in a browser or in various integrated development environments (IDEs) for Python, which offer additional features like code-completion. However, this stack has major problems:

- **Python DS libraries**: Despite their huge application programming interfaces (APIs), knowledge of a single state-of-the-art Python libraries is insufficient to write complete DS pipelines because each library focuses solely on a subtask like the preparation of tabular data, plotting, or classical machine learning. As the libraries are developed largely independently, clients must deal with
  - *overlapping functionality*, which further increases the combined API size and hinders discovering the functionality needed for a task,
  - *differing design philosophies*, which hinders learning, and
  - *differing levels of abstraction*, which makes the interaction between libraries difficult. Clients must write lengthy *glue code* to convert the output of one library into the expected format of another library.

  The conversion between libraries also often loses information that would be valuable for improved error messages, e.g. the column names of a table. This decreases *safety*, i.e. the ability to prevent or detect errors and to help users recover from them.
- **Python**: Python is a dynamically typed language, so by default even basic type errors are only caught at runtime once the faulty statement gets executed. Executing incorrect programs is always a waste of time, but this particularly applies to long-running DS pipelines. Static type checking of Python code is available via external linters, but they only work if
  - libraries provide static type annotations for parameters and results of functions, which are optional in Python and, thus, often omitted,
  - clients install and configure these linters, and
  - clients write their own code in a way that enables static type checking, e.g. by annotating the types of parameters and results of their own functions.

  The linters cannot check other preconditions, like whether a number is inside some interval, either. Overall, Python is lacking safety.
- **Notebook execution**: The cells and hidden state (a mapping from variables to their last assigned value) of a notebook enable *partial re-execution* of pipeline code after changes. This can accelerate development, as users may skip rerunning expensive operations whose results are still up-to-date.

  However, after code changes, the hidden state may become stale, so users must cautiously rerun a subset of their cells in the right order. Rerunning too few cells or running them in the wrong order leads to subtle bugs. However, if users rerun too many cells for safety, or do not split expensive operations into their own cell, the efficiency gain of notebooks diminishes.

In this thesis, we present an alternative for each layer of the DS development stack:

- **Library**: The *Safe-DS library* is written in Python and offers commonly needed features for DS pipelines behind a *simple*, *integrated*, and *consistent* API. It supports data preparation for tables, time series, and images, plotting, and supervised machine learning (ML), which enables writing entire DS pipelines *without any glue code* to connect different libraries.

For *safety*, preconditions are checked at the earliest opportunity at runtime. The library also has type annotations for parameters and results of functions, which allow static type checkers to catch type errors without running code if clients use those tools and write their own code accordingly.

For fast development and easy maintenance, the Safe-DS library internally uses the original Python libraries for DS, but offers an improved API to clients using the *adapter design pattern*. We can, thus, benefit from the maturity of the original libraries, and consume future bug fixes or performance improvements immediately.

The decision which features to include in the Safe-DS library is supported by the analysis of client code of the original libraries, where we counted how often API elements are used. The preconditions to check are partially derived from the documentation of the original libraries. Further API changes, like renaming or moving an API element, were done in a custom graphical user interface (GUI). Adapter code was then generated automatically. This entire process is applicable to any scenario where a developer wants to provide a new API for a library, and we call it **adaptoring**.

- **Language**: The Safe-DS library is already safer than the original libraries. To further improve safety, we developed the *Safe-DS language*, which consists of a simple, easy-to-learn *pipeline language* to write DS pipelines, and a *stub language* to safely integrate Python libraries. The Safe-DS library is included by default via the stub language.

  Without additional configuration, or any added effort when implementing pipelines, we statically catch *type errors* (e.g. passing a string where an integer is expected), *boundary errors* (e.g. passing the value −1 if it must be non-negative), and *state errors* (e.g. inferring with an untrained ML model). *Schema errors* (e.g. accessing a column that does not exist in a table) are detected early by running minimal code.

- **Execution**: We developed an execution system for programs written in the Safe-DS pipeline language that is *correct* (runs all required code in the right order), and *minimal* (runs only the required code). Instead of coarse, user-defined cells, it is based on the individual operations in the code, such as calls, and automatically derives dependencies between them to compute a suitable execution plan. Thus, regardless of how users structure their code, they always get correctness and optimal performance. Call results that cannot become stale are cached between program runs, so they need not be recomputed when the same call is rerun.

- **IDE**: The Safe-DS IDE brings all components together. It offers comprehensive support for the Safe-DS language, like *continuous static checking* (displaying errors in the code as it is being written), code-completion, tooltips that show documentation, and more. A *graphical view* complements the textual pipeline language. Here, operations are displayed as nodes of a graph, and data flow is visualized by edges between them.

  Safe-DS pipelines can be run inside the IDE with our execution concept, and results are presented in dedicated views. Tables are opened in a custom GUI, the *table explorer*, which displays their data, statistics and quality checks for all columns, and can create plots. The data can be sorted and filtered in the table explorer, too.

Our evaluation indicates that the full Safe-DS stack is considerably more usable, is significantly easier to learn, lets DS novices get a lot more work done on their own, and greatly accelerates development compared to the traditional Python stack.

Acknowledgments

Supervisors. First, I want to thank *Jens* for the opportunity to pursue a PhD and *Prof. Dr. Bauckhage* for the chance to attend the ICSE and SANER conferences in person after the pandemic finally settled down, which was a fantastic experience.

I would also like to express my deep gratitude to *Günter*, for his uplifting support, the insightful discussions, and his guidance to bring all ideas for Safe-DS together. It was great fun working with you!

Contributors. I am sincerely grateful to all contributors to the various projects under the Safe-DS umbrella, particularly all students I helped supervise. Your dedication brought the initial vision and concepts for Safe-DS to life, and I highly enjoyed collaborating with you:

- *Nils* wrote the precondition extractor for the adaptoring approach.
- *Simon*, *Andreas*, and *Alexander* added NNs to the Safe-DS library and enhanced the data containers.
- *Arsam* created an automated generator for Safe-DS stubs.
- *Winston* contributed the code generator and runner for the Safe-DS language.
- *Lukas* proved that purity inference for Python is highly challenging.
- *Bela* implemented an explorer for tabular data.
- *Gideon* worked on a graphical view for Safe-DS pipelines.
- *Camilla* evaluated the Safe-DS library, language, and IDE and compared it to the state of the art.

All works will be cited later at the appropriate places and explained in more detail.

Family. Major thanks to my parents, *Wiltrud* and *Heinz*, for always being supportive of my work, even as delays kept piling up like at the Deutsche Bahn. You made it possible that the train still arrived eventually.

Open-source tools. Safe-DS builds on top of many outstanding open-source tools, like Polars, Matplotlib, scikit-learn, and PyTorch. In particular, I want to thank the Langium team for providing a brilliant framework for language development and answering questions quickly and comprehensively.

Funding. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) under project Simple-ML (01IS18054).

Contents

1 Introduction 1
1.1 Publications 2
1.2 Developed Open-Source Tools 3
I Adaptoring 4
2 State of the Art: API Transformation 5
2.1 Existing Approaches and Tools 5
2.2 Research Gap 7
3 A New Approach: Adaptoring 8
4 Automated API Simplification 8
4.1 Inferring Deletions from Usage Counts 9
4.2 Inferring Deletions from Usefulness 10
4.3 Inferring Parameter Optionality 11
5 Automated Extraction of Preconditions 12
5.1 Supported Preconditions 12
5.2 Precondition Detection 13
5.3 Linking Preconditions to API Elements 14
6 Automated Adapter Generation 14
7 GUI for Manual API Editing 16
8 Handling Evolution of the Original Library 19
9 Implementation (Adaptoring) 19
10 Evaluation (Adaptoring) 23
10.1 Impact of Usage-Based Transformations on API Size 23
10.2 Quality of Extracted Preconditions 25
10.3 Runtime of Automated Steps 26
10.4 Usability of API Editor GUI 26
10.5 Threats to Validity 29
11 Related Work (Adaptoring) 29
12 Limitations and Future Work (Adaptoring) 30
13 Conclusion (Adaptoring) 31
II A Library for Data Science 33
14 State of the Art: Python Libraries for DS 34
14.1 Data Preparation for Tables 36
14.2 Data Preparation for Time Series 37
14.3 Data Preparation for Images 38
14.4 Data Visualization 39
14.5 Classical Machine Learning 40
14.6 Neural Networks 41
14.7 Research Gap 43
15 Goals (Library) 44
16 API Design 45
16.1 General Guidelines 46
16.1.1 Discoverability 46
16.1.2 Safety 47
16.2 Package Structure 48
16.3 Data Preparation for Tables (Including Time Series) 49
16.4 Data Preparation for Images 52
16.5 Data Visualization 52

16.6 Containers for Labeled Data ... 53
16.7 Classical Machine Learning ... 54
16.8 Neural Networks ... 55
17 Implementation (Library) ... 56
17.1 Reusing Existing Libraries ... 56
17.2 Containers for Tabular Data ... 57
17.3 Containers for Images ... 58
18 Evaluation (Library) ... 60
18.1 Informal, Long-Term Usage Experiment ... 60
18.2 Structured Usability Study ... 61
18.3 Threats to Validity ... 64
19 Future Work (Library) ... 65
20 Conclusion (Library) ... 66
III A Language for Data Science ... 67
21 State of the Art: Static Safety for DS ... 68
21.1 Existing Approaches and Tools ... 68
21.2 Research Gap ... 72
22 Approach: A Textual Language for DS with Two Parts ... 73
23 Writing DS Pipelines in the Pipeline Language ... 74
23.1 Concepts and Syntax ... 75
23.2 Expressive Power ... 78
24 Integrating Python Libraries via the Stub Language ... 79
24.1 Concepts and Syntax ... 79
24.2 Mapping Stubs to Python ... 82
24.3 Documentation ... 84
24.4 Automated Stub Generation ... 86
25 Safety ... 87
25.1 Types ... 88
25.1.1 Types for Library Integration ... 88
25.1.2 Types for Intermediate Inference Results ... 90
25.2 Subtype Relation ... 91
25.3 Type Inference ... 96
25.4 Type Checking ... 99
25.5 Constraints ... 99
26 Implementation (Language) ... 102
27 Related Work (Language) ... 103
28 Future Work (Language) ... 104
29 Conclusion (Language) ... 104
IV Executing Data Science Pipelines ... 105
30 State of the Art: Notebook Execution ... 107
30.1 Existing Approaches and Tools ... 108
30.2 Research Gap ... 109
31 General Approach for Partial Execution ... 109
31.1 Fine-Grained Dependency Graph ... 109
31.2 Initial Program Execution ... 111
31.3 Program Re-Execution ... 112
31.4 Freeing Memory ... 114
31.5 Parallel Execution of Operations ... 115

32 Executing Safe-DS Pipelines ... 115
32.1 Manifest Purity in the Stub Language ... 115
32.2 Inferred Purity in the Pipeline Language ... 116
32.3 Code Generator ... 117
32.4 Runner ... 118
32.5 Purity Inference for Python ... 119
33 Implementation (Execution Concept) ... 120
34 Evaluation (Execution Concept) ... 121
34.1 Effect of Fine-Grained Dependency Graphs ... 122
34.2 Effect of Memoization ... 123
34.3 Effect of Strategies for Freeing Memory ... 124
35 Related Work (Execution Concept) ... 125
36 Future Work (Execution Concept) ... 125
37 Conclusion (Execution Concept) ... 126
V An Integrated Development Environment for Data Science ... 128
38 State of the Art: Development Tools for DS ... 129
38.1 Generic Python IDEs ... 130
38.2 Graphical No-Code Tools ... 130
38.3 Value Inspection ... 131
38.4 Research Gap ... 133
39 The Safe-DS Approach ... 133
39.1 Textual IDE ... 135
39.2 Graphical View ... 138
39.2.1 Syntax ... 139
39.2.2 Editing and Value Inspection ... 142
39.3 Table Explorer ... 143
40 Implementation (IDE) ... 145
41 Evaluation (IDE) ... 146
41.1 Bachelor Lab ... 147
41.2 Medical Workshop ... 147
41.3 Structured Usability Study ... 148
41.4 Discussion ... 150
42 Future Work (IDE) ... 151
43 Conclusion (IDE) ... 151
Conclusion ... 153
Bibliography ... 156
Figures ... 170
Listings ... 175
Tables ... 177

1 Introduction

Data science (*DS*) deals with the extraction of generalizable knowledge from structured data (e.g. tables) and unstructured data (e.g. text or images) [1]. The field has consistently gained importance in the past years due to the rapid increase in data produced worldwide, from an estimated 33 Zettabyte[1] in 2018 to a projected 175 Zettabyte in 2025 [2].

DS process. The *DS process*[2] describes the stages a data scientist must consider for knowledge extraction. The process differs between authors and teams, but typically has at least the following stages [3]:

1. *Data acquisition*: Data is collected from relevant sources.
2. *Data preparation*: The raw data is cleaned, filtered, and organized.
3. *Feature engineering*: Suitable features for model training (e.g. columns of a table) are selected or created from the data. If the available data is deemed insufficient or of too low quality, the data scientist goes back to a previous step.
4. *Modeling*: A model is created to analyze the data. This stage uses techniques from statistics or machine learning (ML).
5. *Training*: The model is trained on a portion of the available data (*training data*).
6. *Evaluation*: The model predicts the target value on data it has not seen during training (*validation data*). If its performance according to some metric is insufficient, the data scientist goes back to step 3 or 4.
7. *Prediction*: The model predicts the target value on data it has never seen previously (*test data*) and its performance is evaluated according to some metric. This provides a measure for the expected performance in production.

Afterward, the created model may be *deployed* somewhere, so it can be used for continuous prediction. This stage is out of the scope of this work, though. The DS process is mostly sequential, but there are *feedback loops* back to previous steps, as explained above and visualized by Figure 1.

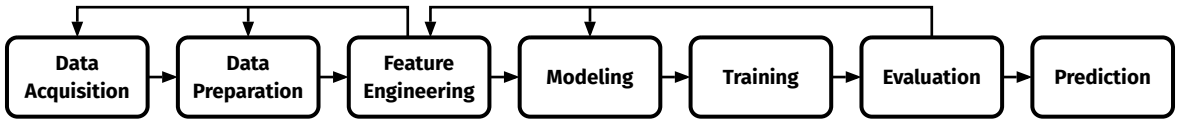


Figure 1: The DS process is mostly linear but has built-in feedback loops [3].

DS pipelines. During the DS process, a data scientist may create a *DS program* to manifest all decisions taken. The DS program is run to inform whether backtracking through a feedback loop is required. After backtracking, the DS program is updated to incorporate any further decisions by the data scientist. Unlike the DS process, the DS program is typically purely sequential [3], as it describes only the current result of the iterative DS process. Due to the sequential nature of DS programs, we will henceforth refer to them as *DS pipelines*.

The need for safety. Due to the feedback loops, DS pipelines are executed many times during the DS process and a single execution may take between minutes to several hours, depending on the size of the data. Hence, it is crucial to avoid runtime errors, which would waste development and computing time, and ultimately money. Thus, the main focus of this thesis is on providing *safety* for the development of DS pipelines, which we define as the prevention or detection of errors and aiding users to correct them.

[1] One Zettabyte is a billion Terabyte.

[2] This is also called DS workflow, DS lifecycle, or DS pipeline in the literature.

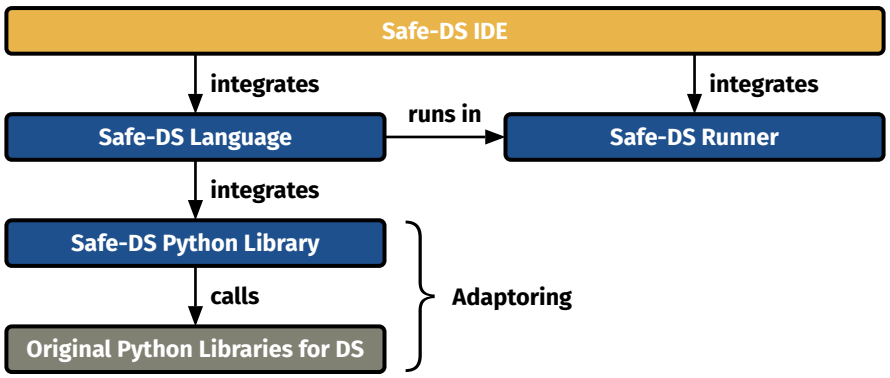


Figure 2: The complete Safe-DS stack. Adaptoring helped create the Safe-DS library.

The Safe-DS stack. We introduce a set of tools, which we call the Safe-DS stack (summarized in Figure 2), to ease the development of DS pipelines and provide the desired safety:

- **Part I** covers the process of *adaptoring* to quickly develop an alternative application programming interface (API) for an existing library without code duplication. Adaptoring can hide useless API elements and improve the design of the remaining ones to ease learning the API and prevent misuse. We applied adaptoring to Python libraries for DS.
- **Part II** discusses our user-friendly, integrated Python library for the implementation of complete DS pipelines. It checks preconditions early at runtime and provides type annotations that can be checked statically by external linters. The consistent, high level of abstraction of its API (e.g. classes for tables and images) provides a close mapping to the users' domain model to ease learning. Any required conversion, e.g. to tensors, is handled in the background, which greatly reduces the code that must be written and, hence, the room for errors.
- **Part III** investigates our language for DS with sophisticated, built-in static checking for DS pipelines developed in it. It can statically detect type errors, boundary errors (a number is outside the legal interval), and state errors (e.g. inferring with an untrained ML model). The language greatly improves safety without any negative impact on the effort to write DS pipelines.
- **Part IV** describes how we correctly and efficiently run DS pipelines written in our language. It avoids the pitfalls of the cell concept of notebooks, where users can introduce subtle bugs by forgetting to execute cells or running them in the wrong order.
- **Part V** presents an integrated development environment (IDE) to develop and run DS pipelines written in our language. It displays any errors directly in the code as it is being written and shows users the available API elements in a code-completion dialog.

Thesis structure. Due to the breadth of the covered topics, each component is explained in a separate part with its own abstract, introduction, survey of the state of the art, description of the approach, discussion of the implementation, evaluation (where applicable), and conclusion. Chapters are numbered consecutively across parts to reduce nesting in cross-references. The thesis closes with a conclusion that summarizes the results of all parts.

1.1 Publications

This thesis is based on several peer-reviewed papers I published previously while working toward my PhD. My mentor, Dr. Kniesel-Wünsche, helped refine the ideas leading up to the papers and assisted with the presentation of the papers' contents. The lists below contain my papers chronologically.

The content of this thesis is completely rewritten and restructured to improve coherence between the topics, though. Each thesis part references the papers it is based on in a box at the top of its first page.

- Peer-reviewed conference papers:
  - **Lars Reimann** and Günter Kniesel-Wünsche, "Improving the Learnability of Machine Learning APIs by Semi-Automated API Wrapping," in *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results ICSE (NIER) 2022, Pittsburgh, PA, USA*, 2022, pp. 46–50, doi: 10.1109/ICSE-NIER55298.2022.9793507
  - **Lars Reimann** and Günter Kniesel-Wünsche, "An Alternative to Cells for Selective Execution of Data Science Pipelines," in *45th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE, Melbourne, Australia*, 2023, pp. 129–134, doi: 10.1109/ICSE-NIER58687.2023.00029
  - **Lars Reimann** and Günter Kniesel-Wünsche, "Safe-DS: A Domain Specific Language to Make Data Science Safe," in *45th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE, Melbourne, Australia*, 2023, pp. 72–77, doi: 10.1109/ICSE-NIER58687.2023.00019
  - **Lars Reimann** and Günter Kniesel-Wünsche, "Adaptoring: Adapter Generation to Provide an Alternative API for a Library," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland*, 2024, pp. 192–203, doi: 10.1109/SANER60148.2024.00027
- Peer-reviewed workshop papers:
  - **Lars Reimann** and Günter Kniesel-Wünsche, "Achieving Guidance in Applied Machine Learning through Software Engineering Techniques," in *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal*, 2020, pp. 7–12, doi: 10.1145/3397537.3397552

1.2 Developed Open-Source Tools

All tools I created during my PhD are open-source and available on GitHub under the Safe-DS organization[3]. The following list orders them by importance and refers to the thesis parts where they are introduced:

- **DSL**: A domain-specific language for DS that catches errors in pipelines without running them (Part III). This repository also includes an extension for Visual Studio Code that offers comprehensive language-specific features, like code-completion (Part V).
- **Library**: A user-friendly, integrated Python library for DS (Part II).
- **Runner**: A server to run programs written in our language that were compiled to Python (Part IV).
- **Stub-Generator**: A command-line tool to integrate Python libraries into the language (Part III).
- **Library-Analyzer**: A command-line tool to analyze Python libraries and their client code (Part I).
- **API-Editor**: A graphical user interface to alter the API of a Python library (Part I).

Most of the code was written by me, but several students contributed significantly to these tools as part of their Bachelor's theses and will be cited appropriately in the relevant thesis parts. In those cases, I helped with their supervision by answering any technical questions, and doing code reviews.

[3] https://github.com/Safe-DS

I Adaptoring

This part is a rewritten and extended version of two papers published previously as part of my PhD thesis:

- *Improving the Learnability of Machine Learning APIs by Semi-Automated API Wrapping* [4]
- *Adaptoring: Adapter Generation to Provide an Alternative API for a Library* [5]

Abstract. Third-party libraries are key to fast application development, since they provide ready-made solutions to recurring tasks. To enable easy learning and use, libraries must provide a well-designed *application programming interface* (API). An obscure API instead slows the learning process and can lead to erroneous use.

The usual approach to improve the API of a library is to edit its code directly, either keeping the old API but deprecating it (temporarily increasing the API size) or dropping it (introducing breaking changes). If the library's maintainers are unwilling to make such changes, others need to create a copy of the library, which they can change. But then it is difficult to incorporate implementation changes to the original library, such as bug fixes or performance improvements.

In this thesis part, we instead explore the use of the adapter design pattern to provide a new API as a new library that calls the original library internally. This allows the new library to leverage all implementation changes to the original library, at no additional cost. We call this approach *adaptoring*.

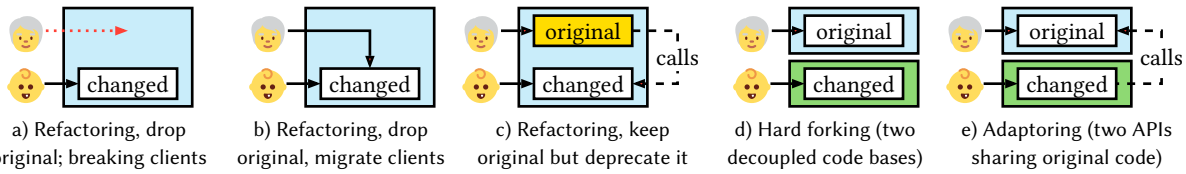
To make the approach practical, we identify API transformations for which adapter code can be generated automatically, and investigate which transformations can be inferred automatically, based on the usage patterns and documentation of the original library. For cases where automated inference is not possible, we present a tool that lets developers manually specify API transformations, while still benefitting from automated generation of adapter code. Finally, we consider the issue of migrating the generated adapters if the *original* library introduces breaking changes.

We implemented our approach for Python, demonstrating its effectiveness to quickly provide an alternative API, even for large libraries.

Nowadays, code is rarely written fully from scratch but is instead based on third-party libraries. Application developers can focus on business logic and no longer need to reinvent the wheel to solve recurring tasks, e.g. to implement a web server. This drastically accelerates application development. Moreover, popular libraries are publicly scrutinized to ensure they work properly, which is crucial for encryption libraries, for instance. Data science (DS) is no exception, and many excellent libraries exist already for all stages of the DS process (Chapter 14).

Application developers interact with a library via its *application programming interface* (API), a collection of *API elements* like classes, fields, functions[4], and parameters. The API must, thus, be easily learnable and usable [6]–[8]. Unfortunately, third-party libraries are often developed by many contributors over a long time, so their APIs can be inconsistent, complex, and suffer from design flaws [9], [10]. The inconsistency is emphasized further if the host language gradually adds new features, which are then used in new parts of the API, but not yet in the old parts of the API.

[4]We subsume global functions, methods, and constructors as functions.



a) Refactoring, drop original; breaking clients

b) Refactoring, drop original, migrate clients

c) Refactoring, keep original but deprecate it

d) Hard forking (two decoupled code bases)

e) Adaptoring (two APIs sharing original code)

Figure 3: The discussed approaches to create a new API for a library. Blue boxes stand for the original library and green boxes for the new one. Original users (👶) interact with the original function until they migrate their code. New users (👶) directly use the changed function. Adaptoring builds the new API based on the unmodified, original one.

2 State of the Art: API Transformation

The maintainers of a library can improve its API in different ways, summarized in Figure 3 and further discussed below.

2.1 Existing Approaches and Tools

API refactoring. The obvious path towards an improved API is to change the original library directly. This can be done systematically using *refactorings*, which are step-by-step guides to change how code is structured without changing its meaning.

Fowler [11] dedicates an entire chapter to API refactorings, but basic refactorings, like renaming a public function, can also affect the API. Dig and Johnson [12] analyzed breaking API changes to four libraries and discovered that 81%–97% changed code structure but preserved code semantics, i.e. were refactorings. IDEs like Eclipse[5] or IntelliJ[6] provide various automated refactorings that can be used to alter an API. For broader coverage of the vast literature on refactoring, we refer to previous surveys [13]–[15].

However, *client code*, i.e. code that uses the API, may break after API refactorings. Since existing client code is generally unknown, and therefore cannot be updated consistently by the library developers, the usefulness of automated refactorings is typically limited to updating the library's own tests and examples.

Migration of client code. Updating unknown client code after breaking changes to a library requires specialized tools, to avoid tedious and error-prone manual migration:

1. Henkel and Diwan [16] capture refactorings during library development, which are then replayed on client code. Their approach requires use of the Eclipse plugin CatchUp! by library maintainers and clients.
2. Dig et al. [17] developed a tool that can correctly detect more than 85% of the refactorings applied to a library. It produces less than 10% false positives.
3. The APIEvolutionMiner [18] detects added and deleted calls between two revisions of a library. From this, it determines suitable replacement rules for calls to removed methods. Such rules can also be specified manually. The tool cannot automatically apply these rules to client code.
4. Diff-CatchUp [19] first determines API changes between two versions of a library automatically. Client developers then select a compiler error caused by breaking API changes. Afterward, Diff-CatchUp suggests possible migration paths. The tool is implemented as an Eclipse plugin.
5. LibSync [20] derives migration paths between two versions of a library by analyzing client code that was already migrated successfully by hand. The authors report a precision of 100% and a recall of 91% for the tool.

[5] https://www.eclipse.org/topics/ide/
[6] https://www.jetbrains.com/idea/

While these tools help with the migration of client code, none of them fully eliminates migration issues. They either place an additional burden on library maintainers (Tool 1), can only handle a subset of possible breaking changes (Tools 1–4), or require access to client code that was already migrated (Tool 5). Moreover, Tools 1–4 cannot handle deprecation of methods, so libraries are *forced* to introduce breaking changes, which in turn forces an immediate reaction of clients that want to upgrade to the new version. Tool 5 could potentially handle deprecations, but again only once enough client code was adjusted manually.

Deprecation of API elements. Breaking changes and challenging migration of client code can be deferred indefinitely by keeping the original API element. Typically, it is marked as *deprecated* and its implementation is changed, so it forwards calls to the new API element that replaces it. A deprecation message in the documentation or during runtime alerts users that they must migrate their code, and can provide migration guidance and a deadline when the original API element is removed. Perkins [21] proposes to inline deprecated methods into client code as a simple migration strategy, i.e. replace calls to them with their body, which he found to work in over 75% of the cases.

But Kao et al. [22] stress that "API deprecation entails significant cost to API developers" including "serious threats to security, performance, or code quality [and] the cost of migration guide provision". Moreover, keeping the original API element adds bloat to the API that is irrelevant for any *new* clients, which can further hinder learning and correct use [23].

Finally, some API changes might require additional unwanted changes to use deprecation. For example, swapping two parameters of the same type is only possible if a new function with a different name is introduced, regardless of whether the host language supports overloading or not.

Hard forking. If the maintainers of the library do not want to change the API, external developers must create a new library to provide the original functionality behind the desired API. This can be done by creating a copy of the original library, the *hard fork*, and changing this instead.

However, the fork is decoupled from the original library, so any bug fixes or performance improvements on the original library must be repeated for the fork as well. Maintaining the hard fork is, therefore, time-consuming.

Adapter library. The adapter design pattern [24] can remedy the duplication problems of hard forking: The adapter provides the desired target interface to clients, but internally forwards calls to the original function (Figure 4). Instead of providing the new API as a separate copy of the original library, it can be offered as a new library of adapters. Thus, future implementation changes to any original function are immediately available to the new library, too.

Each API transformation that can be expressed by an adapter can also be achieved via a refactoring that adds the adapter code to the original library and removes the original API element. Similarly, every refactoring, whose implementation only accesses public API elements, can be encapsulated in an adapter.

In the rare cases, where access to private elements of the original library is required, e.g. for *extracting a function* that should become part of the new public API, library maintainers can fall back to the related refactoring. External developers can use adapters as a primary approach and fall back to *selective* hard forking, by copying only elements that are inaccessible otherwise. This still simplifies maintenance, compared to full hard forking.

We only found one work by Jugel [25] that investigated using adapters to improve an API. It allows deleting and renaming API elements of Java libraries in a UML editor and creates adapters automatically.

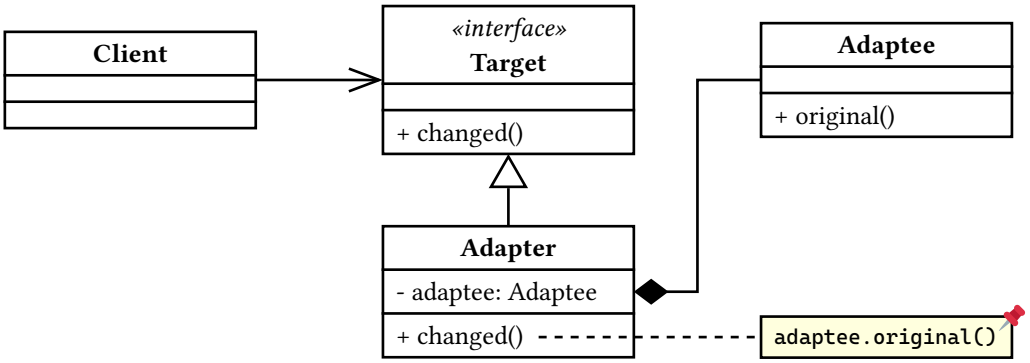


Figure 4: UML class diagram of a variant of the adapter pattern that uses object composition [24]. The client wants to call `Target.changed`, but only `Adaptee.original` exists. The adapter bridges this gap.

2.2 Research Gap

Out of the discussed approaches, only the use of the adapter pattern offers a general solution that can be applied by library maintainers and external developers alike (Table 1):

Adapters for library maintainers. For library maintainers, it offers a means to create a new API without breaking clients or bloating the library's API with deprecated elements. They can use adapters during a large redesign of the API to create a new API that is clearly separated from the original API. New clients can then immediately use the new API, while old clients have time to migrate. After a transition period, the original API can be scrapped and calls to it can be inlined.

Adapters for external developers. External developers can use the adapter approach as an alternative to hard forking, to avoid future maintenance issues. The adapters only need to be updated due to breaking changes to the original API. The adapter layer can be published, so the effort to create it need not be repeated.

Adapters for different user profiles. Furthermore, library maintainers and external developers can easily support different user profiles by providing multiple adapter libraries: For experts, they can offer the full API, while for novices, they can offer a small API that only offers essential functionality. This accelerates application development and prevents misunderstanding the purpose of an overwhelming amount of API elements [26], [27]. Here, the adapters additionally play the role of a facade [24].

Deficiencies of prior work. The only existing work that uses adapters for this purpose [25], however, has shortcomings that limit its practical use:

- Only deletions and renamings are supported as possible API transformations.
- Users must specify transformations manually, which is error-prone and prohibitively tedious for large libraries.

Table 1: Approaches towards a new API for a library.

Criterion	Refactoring + Removal	Refactoring + Migration	Refactoring + Deprecation	Hard Forking	Adapter Library
Usable by library maintainers	✓	✓	✓	✓	✓
Usable by external developers	×	×	×	✓	✓
Backward compatibility	×	✓	✓	✓	✓
Lean API	✓	✓	×	✓	✓
Easy maintenance	✓	✓	✓	×	✓

- The UML editor for the manual specification of transformations has low usability, as reported by the author.
- Evolution of the adapters after breaking changes to the original API is not considered.
- No prototype is publicly available.

Therefore, the remainder of this thesis part explores ways to make an adapter-based solution practical. The proposed approach is called adaptoring.

3 A New Approach: Adaptoring

Adaptoring is a refinement of the adapter-based approach. With adaptoring, the new API is described as a set of API transformations that are applied to the original API. The new API is then implemented as an adapter library, which internally calls the original library. The adapter library is generated automatically. We overcome the limitations of prior adapter-based work by providing the following:

- **Transformation inference** (Chapter 4): We infer deletions, parameter optionality, and suitable default values of optional parameters based on usage data. These transformations *simplify* the API.
- **Automated precondition extraction** (Chapter 5): We analyze the library's documentation to identify preconditions, and check extracted preconditions in the generated adapters, to make the API *safer*.
- **Extended adapter generation** (Chapter 6): We consider API transformations beyond deletions and renamings for adapter generation.
- **User-friendly API editor** (Chapter 7): We offer a custom editor for easy modification of inferred transformations and manual specification of non-inferrable transformations.
- **Evolution of adapters** (Chapter 8): We address updates to the generated adapter library after breaking changes to the original library.
- **Publicly available implementation** (Chapter 9): We implemented a prototype of the adaptoring approach for Python libraries.

Our work on adaptoring was mainly motivated by the desire to explore how far we can automate the creation of a new, user-friendly Python library for DS that leverages the ones available currently. The library that ultimately resulted from this work is presented in detail in Part II.

As the adapter pattern is language-independent, so is adaptoring. We limit our discussion and examples to Python, though, because it is the implementation language of our library for DS. Python is also the only language supported by our prototype.

4 Automated API Simplification

On the path to a DS API that is easy to learn and use even by novices, API simplification comes first. *API simplification* means removal of API elements while keeping the design and implementation of the remaining API. This has several benefits:

1. It accelerates learning, since there is less to learn.
2. It improves visibility of the remaining API elements [23].
3. It reduces work for subsequent API redesign, since fewer API elements remain.

However, removal of API elements can also drastically limit expressiveness[7] of the API. In this section, we describe an approach on how to decide which parts of the API to keep and which to drop (Section 4.1 and Section 4.2). For parameters, we also outline when parameters should be made optional, so users *can* still set them, but no longer *have to* (Section 4.3). While this does not reduce the API's *physical size*, it reduces its *conceptual size*, since the size of the API that a user must interact with gets smaller.

[7] Throughout the thesis, we will use the term *expressiveness* to denote the ability to express a concept at all, regardless of ease-of-use.

We limit the following discussion to classes, functions, and parameters. The presented approach can be extended canonically to other kinds of API elements, like fields.

The notions that underlie our analyses of an API are relative to the set of analyzed client programs C. Hence, C should be representative of the general use of the API. For popular open-source libraries, a representative set of client programs can be mined from public repositories, for example from GitHub[8]. In the domain of DS, Kaggle[9] is another source of typical API usages.

4.1 Inferring Deletions from Usage Counts

Usage analysis is our first step towards empirically validated API simplification. It takes as input a set of client programs that use the API and then computes the usage count for each API element:

> **Definition 1** (Usage count): Given a set of client programs C, the *usage count* of an API element e is the sum of all usages of e in C, where
>
> - a usage of a class c is a call to a method of c, including constructors,
> - a usage of a function f is a call of f,
> - a usage of a parameter p is the explicit passing of an argument to p when the containing function is invoked[10].
>
> We denote the usage count of e in C as $\text{usages}_C(e)$. The subscript C may be omitted if there is no ambiguity.

Because Definition 1 excludes fields, the usage count of a class is simply the sum of the usage counts of its methods and constructors. We did not consider usages of classes in type annotations (e.g. to denote the type of a parameter), since they are optional in Python and are ignored by the Python interpreter [28]. The client programs we analyzed (Section 10.1) to determine which API elements to include in our DS library (Part II) hardly use type annotations at all. In other scenarios and especially in statically typed languages, the use of a class in a type annotation should be counted, however.

Usage counts of *optional* parameters may be lower than usage counts of the containing function. If they are not set explicitly, their default value is used implicitly, which is *not* counted.

API elements with a usage count of 0 are *unused* and can be removed without breaking any *analyzed* client program. But as Hyrum's law [29] suggests, it is likely that *other* client programs use these API elements, unless *all* client programs ever written were analyzed. Hence, the removal of unused API elements would still prevent some existing clients from migrating to the adapter library. Because of this, we generalize the notion of unused API elements to arbitrary thresholds:

> **Definition 2** (Rarely used): Given a threshold $T \geq 1$, we call an API element e *rarely used*, if $\text{usages}(e) < T$.

We can now remove rarely used API elements for arbitrary thresholds to further reduce the API size. The threshold is used to balance between an expressive API (low T) and a small API (high T). The special case $T = 1$ removes only unused API elements.

Additionally, different thresholds can be used to create versions of an API for users of different skill levels [30]. Novices are introduced to a strongly reduced API, whereas more experienced users work with a larger API that is closer to the original one.

[8] https://github.com
[9] https://kaggle.com
[10] In contrast, the default value is passed *implicitly*, if optional parameters are not mentioned in the invocation.

Note that for any class c, function f, and parameter p, where c contains f and f contains p, the following relations hold:

$$\text{usages}(c) \geq \text{usages}(f) \geq \text{usages}(p)$$

This follows directly from the definition of the usage count. Hence, for a fixed threshold, the removal of a rarely used class, only removes rarely used functions, and the removal of a rarely used function only removes rarely used parameters.

4.2 Inferring Deletions from Usefulness

Usefulness analysis provides additional hints on which parameters to remove. It takes as input a set of client programs that use the API and computes the number of occurrences of parameter values. The usefulness is then derived from the occurrence counts:

Definition 3 (Occurrence count): Given a set of client programs C, the *occurrence count* of the value v for parameter p reflects how often p is set to v in C. Here, we do not differentiate whether p is set explicitly to v in a call, or whether p is omitted and v is its default value that is used implicitly.

We denote the occurrence count of the value v for parameter p in C as $\text{occurrences}_C(p, v)$. The subscript C may be omitted if there is no ambiguity.

Definition 4 (Usefulness): Given a set of client programs C, the usefulness of a parameter p of some function f is the number of times it is set explicitly or implicitly to a value other than the most common value it takes:

$$\text{usefulness}_C(p) = \text{usages}_C(f) - \max_v(\text{occurrences}_C(p, v))$$

The subscript C may be omitted if there is no ambiguity.

Parameters with usefulness 0 are always set to the same value. Such parameters are *useless* for the analyzed client programs. They can be deleted from the containing function, and references to them in the function's body can be replaced by their unique value. This idea can again be generalized:

Definition 5 (Almost useless): Given a threshold $T \geq 1$, we call a parameter p *almost useless*, if $\text{usefulness}(p) < T$.

We can remove almost useless parameters for arbitrary thresholds to further reduce the API size. The special case $T = 1$ removes only useless parameters. Regarding the choice of the threshold, the same considerations apply as for deletion by usage count.

Theorem 1 (Usage Count vs. Usefulness): Note that for any parameter p the following relation holds:

$$\text{usages}(p) \geq \text{usefulness}(p)$$

Because of this, for a fixed threshold, a rarely used parameter is also almost useless. Vice versa, removing all almost useless parameters, also removes all rarely used parameters, leading to a greater simplification.

Proof: Let f be the function that contains p and $\text{implicit\_count}(p)$ denote how often p is set implicitly to its default value when f is called. For required parameters, $\text{implicit\_count}(p) = 0$. Then by definition

$$\text{usages}(p) = \text{usages}(f) - \text{implicit\_count}(p)$$

Substituting $\text{usages}(p)$ by the above observation and $\text{usefulness}(p)$ by the definition of usefulness in the desired relation, we get

$$\begin{aligned} & \text{usages}(p) && \geq \text{usefulness}(p) \\ \Leftrightarrow\ & \text{usages}(f) - \text{implicit\_count}(p) && \geq \text{usages}(f) - \max_v(\text{occurrences}(p, v)) \\ \Leftrightarrow\ & \max_v(\text{occurrences}(p, v)) && \geq \text{implicit\_count}(p) \end{aligned}$$

This is true, since the maximum occurrence count of a value must be greater than or equal to how often the default value is used implicitly. ■

4.3 Inferring Parameter Optionality

Deleting API elements is a drastic step. For parameters, many programming languages also allow specifying default values, which are used if clients do not set the parameter explicitly when calling the containing function.

Clients can ignore optional parameters, while still having the option to overwrite default values as needed. This simplifies the API and can accelerate development, since they no longer have to understand the purpose of these parameters and how to set them properly (e.g. the learning rate of various ML models). Particularly, frequent context switches to check the documentation are eliminated. However, clients may also forget to overwrite unsuitable default values, leading to decreased safety. Hence, the choice of parameter optionality is a tradeoff between simplicity (optional) and safety (required).

Intuitively, a parameter should only be made optional, if there is a clear default value. Some boolean parameter, that must be `True` and `False` about equally often, should be required instead, so clients must choose carefully themselves.

Our formalized approach uses the empirical usage data we described in Section 4.2: We do a *null-hypothesis significance test* [31, p. 923] considering only the two most common values of a parameter p. Let v_1 denote the most common value, v_2 the second most common value, c_1 and c_2 their respective occurrence counts, and $n := c_1 + c_2$. If the parameter is always set to the same value (useless), then $v_2 := 0$ and $c_2 := 0$.

We now assume the null hypothesis H_0 that in the n cases where v_1 or v_2 were used, they were chosen equally often. If we can reject H_0, we make the parameter optional, with v_1 as the default. Otherwise, we make the parameter required. We reject H_0 if its p-value [32] is less than or equal to some significance level α in a two-sided test. Since H_0 is governed by a binomial distribution [31, p. 263], this means that we reject H_0 if

$$2 \sum_{i=c_1}^{n} \binom{n}{i} 0.5^n \leq \alpha$$

Similar to the threshold parameter to tune the deletion of API elements in Section 4.1 and Section 4.2, the significance level α can be configured to produce a suitable tradeoff between simplicity and safety. A lower α makes it more difficult to reject H_0, leading to more required parameters (higher safety), while a higher α produces more optional parameters (higher simplicity).

The formula is designed, so parameters in rarely used functions are more likely to be made required, opting for safety and preventing breaking changes later to remove or change default values. Since these functions are seldom used anyway, simplicity is barely affected. Once more data is available, these parameters can then be made optional.

5 Automated Extraction of Preconditions

Next, we want to make the remaining API elements safer to use by checking their preconditions early at runtime and failing immediately with helpful error messages if they are violated.

Preconditions can generally be sourced either from code or documentation. If they are already formalized as code, we do not gain much from extracting them and checking them again. Since a failing precondition halts program execution, checking the same precondition a second time, has no noticeable effect[11]. If, however, preconditions are only described using natural language in documentation, implementing them as automated checks can greatly accelerate debugging.

Our approach is, thus, to ignore the existing library code, automatically extract preconditions from library documentation, and implement runtime checks for them in the generated adapters (Chapter 6). This requires

1. to decide which preconditions we initially want to support (Section 5.1),
2. to derive patterns to detect these preconditions in documentation (Section 5.2), and
3. to link the extracted preconditions to API elements (Section 5.3).

5.1 Supported Preconditions

While documentation can contain arbitrary preconditions, automated extraction must be implemented for each kind individually. For a proof-of-concept, we must first decide which kinds of preconditions we want to support. For this, we manually analyzed the documentation of scikit-learn [33], [34] with the help of several students during the lab "Angewandte Softwaretechnologien 2022" and narrowed the list to three common kinds of preconditions:

Literal sets. A parameter may only accept a finite set of literals, typically strings or numbers. For example, the `penalty` parameter of a `LinearSVC`[12] only accepts the strings `'l1'` and `'l2'`. In this case, the documentation of scikit-learn is fairly standardized, expressing the legal literals using set notation, such as `{'l1', 'l2'}`.

Boundaries. Values of a numeric parameter may need to be in some open, closed or mixed interval. For instance, the regularization parameter `C` of an `SVC` has to be positive. Boundaries can be expressed by many different wordings, as shown in Table 2.

Table 2: Wordings for boundaries from the documentation of scikit-learn.

Category	Example	Number of Occurrences
Interval syntax	in [0, 1]	108
Textual interval	between 0 and 1	50
Type with adjective	non-negative float	24
Textual comparison	at least 1	19
Name with comparison syntax	x >= 1	16
Type with comparison syntax	float >= 0	13

[11]Extracting preconditions from code to normalize error messages would still be beneficial, but was not a focus of this work.

[12]SVC means support vector machine for classification.

Table 3: Conditions of parameter dependencies from the documentation of scikit-learn.

Condition	Example	Number of Occurrences
Parameter has exact value	if `shuffle` equals `False`	334
Parameter has value that satisfies relation	if `momentum` > 0	10
Parameter has type	if `n_iter` is an integer	4

Table 4: Actions of parameter dependencies from the documentation of scikit-learn.

Action	Example	Number of Occurrences
Parameter is ignored	ignored by all other kernels	322
Parameter must have value	`stratify` must be `None`	17

Parameter dependencies. The semantics of a parameter may depend on the value of another parameter. The parameter `degree` of an `SVC`, for instance, sets the degree of a polynomial kernel. If the `kernel` parameter is set to anything else, `degree` is silently ignored. This category has the most diverse wordings, since conditions (when does a parameter depend on another; Table 3) and actions (what happens if the condition is met; Table 4) may differ independently.

5.2 Precondition Detection

Next, we discuss how to detect literal sets, boundaries, and parameter dependencies in the documentation: For literal sets, simple regular expressions suffice, since literal sets have a fixed format and neither the curly braces of the set notation nor the quotes of strings are nested.

Regular expressions can also be used to detect boundaries, each wording becoming one alternative. Here, however, preprocessing of the documentation, for example to replace consecutive whitespace by a single space, is already needed to keep the regular expression maintainable. Alternatively, a tokenizer [35, pp. 39-54] can first convert the detailed character stream into an abstract stream of tokens for identifiers, numbers, comparison operators, etc. Parsing can then be implemented on top of the token stream, e.g. as a recursive descent parser [35, pp. 84-89].

For parameter dependencies, regular expressions or a custom tokenizer and parser are not suitable. To prevent false positives, we must ensure that conditions and actions belong to each other. This requires analysis of linguistic features of the documentation: Words are first grouped by their *part of speech* (also called *word class*), which captures their syntactic role in a sentence, like noun, verb, or adjective [36]. Then, relations between words inside a clause or sentence [37] are derived. These relations are often visualized as a dependency tree[13]. The tree for the sentence “This is ignored if verbose equals 0” is shown in Figure 5, including part of speech tags.

Since a clause must contain a verb [37], these are used as the root of the dependency tree and of subtrees. In relation to the root “ignored”, “This” is the passive subject, and “is” is a passive auxiliary verb. “equals” is the root of an adverbial clause and, thus, classified as an adverbial clause modifier. In relation to the root of the adverbial clause, “if” is the marker that introduces the clause, “verbose” is the nominal subject, and “0” the direct object.

Libraries for *natural language processing* (NLP) provide this classification of words by part of speech and the creation of dependency trees. The tree in Figure 5, for example, was created using spaCy [38]. They also offer other features, like tokenization of a sentence into words, or *lemmatization* to reduce words to their root form. All this allows the specification of exact detection rules for parameter dependencies.

[13] Not to be confused with *parameter* dependencies.

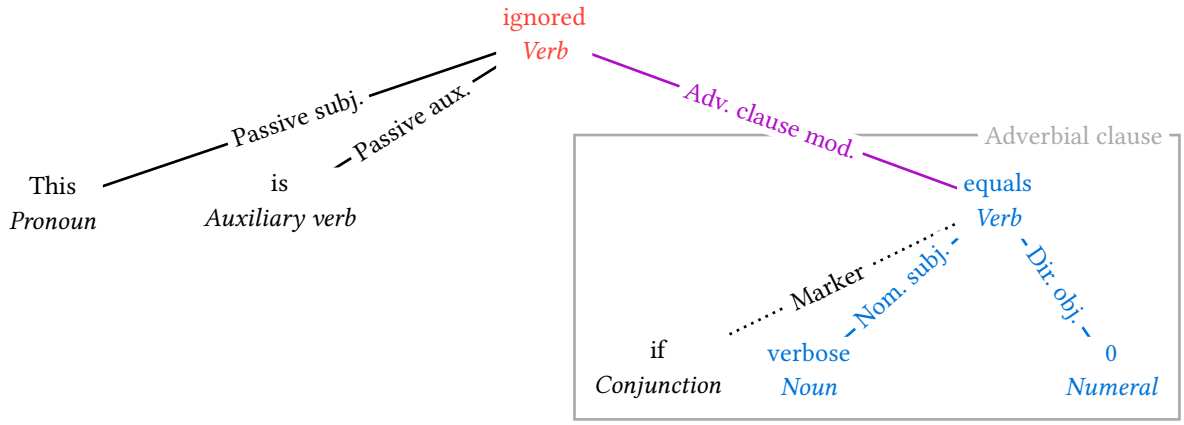


Figure 5: Part of speech tags and dependency tree for a parameter dependency (classified by spaCy). The action is marked in red, the condition in blue. Action and condition are linked by a dependency of type adverbial clause modifier (purple).

For the example from Figure 5, where the action is *Parameter is ignored*, we can look for

- a word with the lemma "ignore",
- a dependency of type adverbial clause modifier starting at "ignore", and
- ensure that the marker of the adverbial clause is "if".

The condition is then everything in the adverbial clause without the marker "if". It can then be further analyzed using NLP techniques to classify it according to Table 3.

5.3 Linking Preconditions to API Elements

We can now detect preconditions in arbitrary text. However, we must still link them to API elements to check them automatically. For this, we use the fact that documentation is not arbitrary text, but typically has a fixed structure.

In the case of Python, there is the additional problem that there is not one predominant standard for documentation (called *docstrings* here) but four, namely NumPyDoc [39], Google's style [40], ReStructuredText [41], and Epytext [42]. Examples for them are shown in Listing 1.

As can be seen in Listing 1, the syntax inside the docstrings is vastly different: NumPyDoc and Google's style break docstrings into sections (`Parameters` and `Args` respectively) and have one entry per parameter, containing its type and a description. ReStructuredText and Epytext do not have sections and use two entries for a parameter's description and its type.

Nevertheless, the information inside the docstrings is the same, regardless of the chosen documentation format. In Listing 1, there is always

- a general description for the function,
- a description for the parameter `n`, and
- the type of the parameter `n`.

Because of this, our approach is to first convert the docstrings into a unified intermediate format. Then, the detection described in Section 5.2 can be run on each piece of the documentation, like a parameter description, individually. Since the documentation is linked to API elements, so are the extracted preconditions.

6 Automated Adapter Generation

So far, we have inferred potential API simplifications and extracted preconditions to check. In the following sections, we refer to such formally described changes to an API as *transformations*. Next, we

```python
def increment_numpydoc(n):
    """
    Increment a number.

    Parameters
    ----------
    n: int
        The number to increment.
    """
```

a) NumPyDoc

```python
def increment_google_style(n):
    """
    Increment a number.

    Args:
    n (int): The number to increment.
    """
```

b) Google's style

```python
def increment_re_structured_text(n):
    """
    Increment a number.

    :param n: The number to increment.
    :type n: int
    """
```

c) ReStructuredText

```python
def increment_epytext(n):
    """
    Increment a number.

    @param n: The number to increment.
    @type n: int
    """
```

d) Epytext

Listing 1: Python docstrings with a general description, a parameter description, and a parameter type in popular formats.

must create adapters to implement these transformations. We explain our approach using the function in Listing 2 as a running example, which is supposed to create a JSON file containing the given data at the given path. Clients can furthermore optionally configure the character encoding, and indentation for composite data structures.

Say, the computed value occurrences (Section 4.2) show that the default value `3` for the `indent` parameter is almost always overwritten and that the values `2` and `4` are used about equally often. The analysis from Section 4.3, thus, suggests making the parameter required instead. This is the transformation, we want to apply.

Trivial adapters. We begin by creating a trivial adapter that provides the same interface as the original function and just passes any arguments along (Listing 3). The adapter also has the same docstring as the original function, so users can easily read it without referring to the original function. This is omitted from Listing 3 for brevity.

Note that since trivial adapters are created for all functions of the original library, they will be included in the output, even if they are unchanged. This is deliberate, so clients no longer have to interact with the original library.

Applying transformations. Now, all transformations that were inferred for the function are applied one after another. Here, it is only the deletion of the default value of `indent`, which yields the adapter shown in Listing 4.

```python
def write_json_file(path, data, encoding="utf-8", indent=3):
    ...
```

Listing 2: The signature of the original function. Its implementation is omitted.

```python
def write_json_file(path, data, encoding="utf-8", indent=3):
    original.write_json_file(path, data, encoding, indent)
```

Listing 3: Initial, trivial adapter with the original signature. The docstring is omitted.

```
def write_json_file(path, data, encoding="utf-8", indent):
    original.write_json_file(path, data, encoding, indent)
```

Listing 4: Adapter after removing the default value of `indent`. Note that the code is invalid, since the optional parameter `encoding` occurs before the required parameter `indent`. This is fixed next.

Table 5: Application of inferred transformations on adapters.

Transformation	Application
Delete class/function	Delete the entire class/function.
Delete parameter	1. Delete the parameter from the parameter list of the containing function. 2. Delete the parameter documentation from the docstring of the containing function. 3. Replace references to the parameter by its most common value.
Make parameter required	Remove the default value of the parameter.
Set default value of parameter	Set the default value of the parameter to the new value.
Check precondition	Add a conditional statement that raises an error if the check fails at the top of the body of the function.

All other transformations we discussed so far are generally also straightforward to apply, as summarized in Table 5. During the deletion of a parameter, we also update the docstring of the containing function to remove the documentation for the removed parameter.

Postprocessing. The adapter may no longer be legal after applying the transformations. In Python, for example, all required parameters must be listed before optional ones [43]. The state shown in Listing 4 violates this rule. We solve this by postprocessing after all transformations were applied. During postprocessing, we place all required parameters first in the order that they appeared previously. They are followed by the optional parameters, again keeping their original relative order. We also remove empty modules, so no empty files are created. The final result is shown in Listing 5. Only this final result is included in the output of the adapter generator.

Note that other transformations may void the need for reordering of parameters. If `encoding` was also made required, the original relative order of `encoding` and `indent` could be kept. Thus, doing the reordering in a postprocessing step instead of after each transformation minimizes changes.

7 GUI for Manual API Editing

The adapter generator explained in Chapter 6 needs transformations as input. It does not matter how these transformations were created. Because of this, we can allow users to (1) review and improve automatically inferred transformations and (2) manually add transformations that cannot be inferred automatically.

To make this review and extension process user-friendly, we developed a graphical user interface (GUI), the API Editor[14], that integrates all required information. It lists API elements of a library, together with their usage data, documentation, and inferred or manually added transformations. The inclusion of documentation prevents time-consuming context switches to online documentation. A screenshot of the GUI is presented in Figure 6. Some GUI elements were omitted for brevity.

```
def write_json_file(path, data, indent, encoding="utf-8"):
    original.write_json_file(path, data, encoding, indent)
```

Listing 5: Adapter after reordering parameters during postprocessing.

[14]https://github.com/Safe-DS/API-Editor

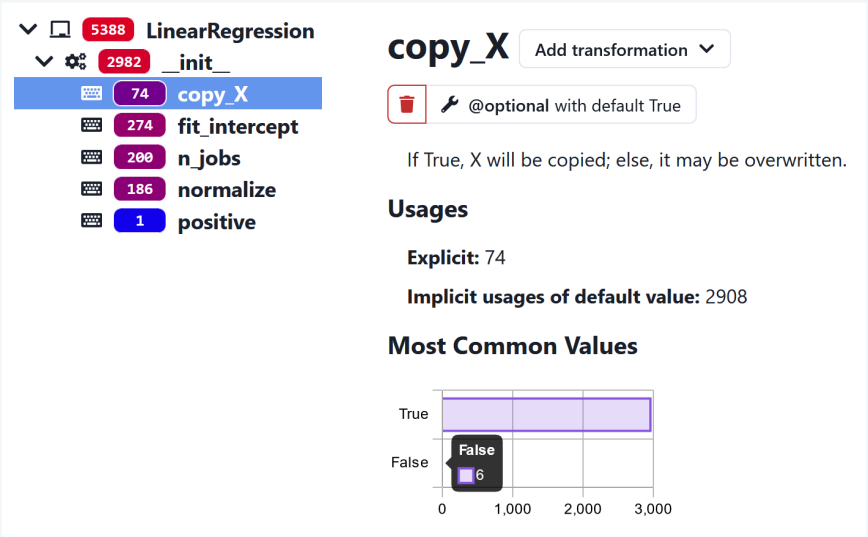


Figure 6: An excerpt of the GUI for review and extension of API transformations. Users can navigate the API elements using the *tree view* on the left and then inspect information about them in the selection view on the right.

Navigation. API elements are displayed in a nested *tree view*, containing packages, classes, functions, and parameters, as seen on the left of Figure 6. Users can optionally display usage count (done here), usefulness, or transformation count in the tree view. Counters are colored from blue (low) to red (high) to help quickly identify hotspots. The tree view can be sorted by these metrics, or alphabetically (done here). By default, parameters are in declaration order, and other API elements are in alphabetical order.

Filtering. Moreover, expressive filters can be applied to focus on specific parts of the API. For example, the filter `usefulness:>20` only keeps API elements with a usefulness greater than 20. The filter `qname:/linear/` keeps only API elements with the string "linear" in its qualified name. Arbitrary regular expressions can be used between the slashes of the `qname` filter. Individual filters can be negated by prepending an exclamation mark. If several filters are used, they must all apply.

Details about an API element. Clicking on an entry in the tree view opens the *selection view* for this API element (right side of Figure 6). From top to bottom, it shows

- the name of the API element and a dropdown to add transformations,
- a list of all current transformations, with buttons to delete and edit them,
- the documentation of the API element,
- usage count and information derived from it,
- a diagram for value occurrence counts that shows exact values on hover (only for parameters).

Transformation editing. Transformations can be added or edited in dedicated dialogs, as presented in Figure 7 for the "Set default value of parameter" transformation (shown as `@optional` in Figure 6). Moreover, the GUI supports a batch mode to quickly add a transformation to all API elements that match the current filter.

In addition to the transformations discussed in previous sections, users can

- rename classes, functions, and parameters,
- move classes and global functions to another package, and
- group parameters into a parameter object [11].

A parameter object is simply an object with fields for each wrapped parameter. This way, related values, like a person's first and last name, can be passed around as a single unit. Rename and move

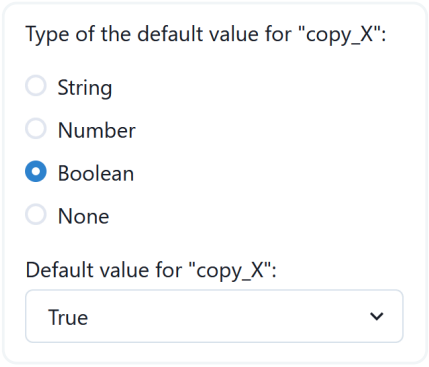


Figure 7: Dialog to add or edit a transformation that sets the default value of a parameter.

Table 6: Application of manually applied transformations on adapters.

Transformation	**Application**
Rename	Set the name.
Move	Move the entire API element to the new module.
Group into parameter object	1. Create a new class for the parameter object with fields for each wrapped parameter. 2. Replace the wrapped parameters in the parameter list of the containing function node with a single new parameter that has the type of the new class. 3. Replace references to wrapped parameters by accesses to the corresponding field of the new parameter.

were chosen because they are common refactorings [11]. Group into parameter object is meant as a proof-of-concept that our approach can also handle more complicated transformations.

During adapter generation, these manually added transformations alter adapters, as discussed in Chapter 6. Their application is summarized in Table 6.

Transformation review. The GUI also has features dedicated to the review of transformations that were created automatically or by a colleague in a collaborative environment (Figure 8): A transformation can be marked as *correct*, *unsure*, or *wrong*. If a transformation is marked as wrong, it is ignored during adapter generation. Compared to outright deleting the transformation, this has the benefit that it prevents accidentally adding the transformation again later. The unsure marker can be used as a reminder to revisit the annotation, or to ask the colleague who created it.

Moreover, entire API elements can be marked as *complete* to indicate that no more transformations are needed. If an API element is marked as complete and all its transformations as correct or wrong, we say that this API element is *done*. The API can be filtered, so only complete or done API elements remain. Naturally, these filters can also be negated.

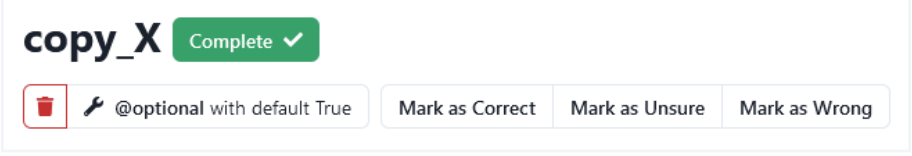


Figure 8: Buttons for transformation review. API elements can be marked as complete, indicating that all transformations are specified. Each transformation can be marked as correct, unsure, or wrong.

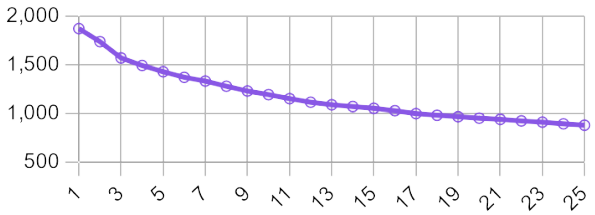


Figure 9: Visualization of the number of remaining parameters for sensible usefulness thresholds.

Visualization for determining thresholds. Finally, the GUI visualizes how varying the usage count/usefulness threshold during API simplification (Section 4.1 and Section 4.2) impacts the size of the remaining API. This helps users choose a suitable threshold. For instance, a user can determine how many parameters remain for which usefulness threshold using the diagram in Figure 9.

8 Handling Evolution of the Original Library

Breaking changes to the original library upon which the generated adapters are based might necessitate updates to the adapters. Newly added API elements can simply be run through the pipeline described in previous chapters to create fresh adapters for them. Likewise, if API elements are deleted, the corresponding adapters can be deleted.

Changes to API elements, for which adapters already exist, are more complicated, however. For example, if a function is renamed in the original library, we *must* change the body of the associated adapter and *might* decide to change its name, too, so it stays synchronized with the original library.

If the developers of the original library and the adapters are identical, standard refactoring tools can be used to always keep the *body* of adapters up-to-date as the original library changes. If the developers differ, the adapters take the role of external client code, so the migration approaches from Chapter 2 are applicable. Given that the number of breaking changes is normally small, these could be handled manually, too. The *interface* of an adapter could also be adjusted manually. Partial automation can streamline this process, though:

1. Record all API transformations required to get from the trivial adapters to the final state (Chapter 6). We refer to this set of transformations as the *adapter branch*, and it is known anyway since it is an integral part of the adapter generation.
2. Derive the API transformations that were applied between the version of the original library for which the adapters were created and the version that should be used as the basis instead. We call this the *maintainer branch*. This task can be handled by diffing tools like UMLDiff [44], DiffCatchUp [45], SemDiff [46], [47], AURA [48], or HiMa [49].
3. Merge the adapter branch and the maintainer branch using approaches for the automated detection and resolution of conflicts, like the ones collected by Mens [50]. Manual conflict resolution may still be needed, e.g. if the same API element has been renamed differently on both branches. This process could be integrated in the GUI for API editing (Chapter 7), where users could either keep all conflicting transformations from the adapter or maintainer branch or decide for each case.

9 Implementation (Adaptoring)

Acknowledgments. The current implementation of the precondition extractor was done by Nils Vollroth in his Bachelor's thesis [51].

Progress and repositories. Apart from the handling of evolution of the original library (Chapter 8), all steps of the adaptoring approach are included in a proof-of-concept implementation for Python. Due to the different programming languages and technologies used, the implementation is split into

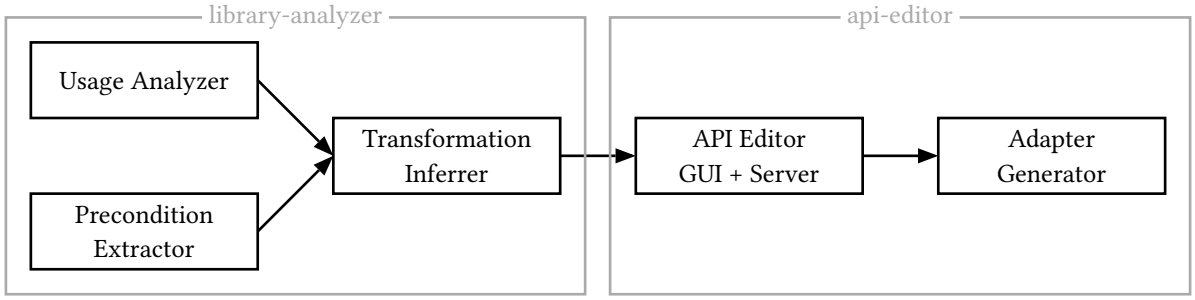


Figure 10: Projects (gray) and components (black) that encompass the proof-of-concept implementation of the adaptoring approach for Python.

two projects: The *library-analyzer* project[15] contains the usage analyzer (Section 4.1 and Section 4.2), precondition extractor (Chapter 5), and an inferrer for deletions (Section 4.1 and Section 4.2), parameter optionality (Section 4.3), and precondition checks (Chapter 5). The *api-editor* project[16] deals with the GUI for manual API editing (Chapter 7), and the subsequent adapter generation (Chapter 6). This structure is summarized in Figure 10.

Usage Analyzer. The usage analysis can either be done statically, by looking only at the source code, or dynamically, by running the downloaded client programs. Static analysis is less exact because (1) call targets might not be known, leading to wrong usage counts, and (2) values might not be known, leading to wrong value occurrence counts for parameters. Examples of these issues are included in Listing 6.

In example a), the receiver `r` has no static type information, which is optional in Python [28], so we only know that the called function is named `action` and that it may be called without parameters. Even with static type information, however, we may not know the exact function that gets called due to dynamic dispatch [52].

In example b), the function `action` either gets imported dynamically from `module0` or `module1`, depending on the parameter `n`. Example c) shows another variant of a dynamic import using Python's `importlib` [53]. Here, the entire module that gets imported is determined by the parameter `name`, providing us only with the name (`action`) and possible arity (0) of the called function. In example d), the value passed to `action` depends on the parameter `n` and is, thus, not known statically.

```python
def untyped_receiver(r):
    r.action()
```

a) Untyped call receiver

```python
def dynamic_import_1(n):
    if n % 2 == 0:
        from module0 import action
    else:
        from module1 import action
    action()
```

b) Dynamic import using statement

```python
def dynamic_import_2(name):
    module = importlib.import_module(name)
    module.action()
```

c) Dynamic import using `importlib`

```python
def computed_value(n):
    action(n % 2 == 0)
```

d) Computed parameter value

Listing 6: Problems with static usage analysis.

[15]https://github.com/Safe-DS/Library-Analyzer
[16]https://github.com/Safe-DS/API-Editor

Dynamic analysis also has major downsides, however: Running random code downloaded from the internet is a huge security risk and requires proper sandboxing. Moreover, dynamic analysis is much more expensive than static analysis, particularly in the DS context, where running *a single client program* can take minutes to several hours. This makes the analysis of thousands of client programs prohibitive.

Because of the lengthy runtime of dynamic analysis, our prototype employs the less exact static analysis. It is implemented in Python and uses the library astroid[17] to build an abstract syntax tree (AST), find calls, and derive call targets — subject to the restrictions explained above, which cannot be circumvented when statically analyzing Python code. Then, usage counts are computed as described in Section 4.1.

When computing value occurrence counts, only literal values, e.g. boolean literals (`True`/`False`), int literals (`1`), float literals (`1.0`), or string literals (`'hello'`) can be considered identical. Other expressions, like references to variables, are all counted as different values, even if they have the same program text. Thus, if some parameter is always set to a reference to some variable `x` in all client programs, it is still not marked as useless, since `x` may represent a different value each time it occurs.

Partial evaluation [54] of expressions is not implemented, so e.g. `2` and `1+1` are regarded as different values. While this would lead to more exact value counts in general, we found little need for it in the DS context, where parameters were usually just set to literals.

Precondition Extractor. The precondition extractor uses the Python library `docstring_parser`[18] to convert docstrings in the formats NumPyDoc, Google's style, ReStructuredText, and Epytext (Listing 1) into a common intermediate format.

Afterward, literal sets are extracted from the type field of the intermediate format (e.g. `{'l1', 'l2'}`). Boundaries and parameter dependencies are extracted with a set of spaCy [38] matching rules from the natural language description of each parameter. spaCy is a Python library for natural language processing. There are also matching rules for literal sets looking for phrases like "possible values are…" in the natural language description, in case a library does not use the set notation in the type field.

Due to lack of training data, the matching rules were created manually. Once sufficient training data is available, e.g. by enriching the results of the current extractor, the rule-based matching could be replaced by an ML model to improve generalization. Furthermore, Wang and Zhao [55] suggest that the inclusion of client code analysis can improve precondition extraction. Since the usage analysis needs client code anyway, this could be another path for future work.

Transformation Inferrer. The inference of deletions and parameter optionality uses the results of the usage analysis and a user-defined threshold, as described in Section 4.1, Section 4.2 and Section 4.3. The extracted preconditions can also be converted to transformations in a straightforward way: For example, if a parameter `ratio` must be between 0 and 1 (inclusive), a suitable runtime check is added before the original function is called.

API Editor GUI. For maximum portability, the API Editor GUI runs in the browser and is implemented in TypeScript[19]. Notably, it uses React[20] for the user interface and Redux[21] for state management. We picked these libraries due to their maturity, popularity, and availability of development tools.

[17] https://github.com/pylint-dev/astroid
[18] https://github.com/rr-/docstring_parser
[19] https://www.typescriptlang.org/
[20] https://react.dev/
[21] https://react-redux.js.org/

After changes, state is immediately persisted into the browser's IndexedDB [56], [57], an object-oriented database system. When the application is started, the last state is restored. Thus, users can close the browser at any point and continue their work later.

The current selection in the tree view is reflected in the browser's address bar and interacts with the browser's history API [58], [59]. This allows users to set bookmarks and use the standard navigation buttons and keybindings of the browser to move to previous selections.

API Editor Server. To access the IndexedDB, the application must be hosted by a server, rather than accessed directly via the file system. The server may be run locally. Our server is implemented in Kotlin[22], a language that runs on the widely available Java Virtual Machine [60]. Among other advantages, Kotlin is safer than Java because it differentiates nullable and non-nullable values, preventing null pointer exceptions [61]. The server uses the Kotlin library Ktor[23].

Adapter Generator. The adapter generation is also implemented in Kotlin and included in the server. Since implementing API transformations as operations on the program text is prone to errors[24], we instead went in a different direction: Adapters are represented as an abstract syntax tree (AST) and transformations are done on this AST. To generate Python code, the nodes of the AST are simply visited in postorder and serialized as a final step. Parents compose the text generated for their children. A parameter node with the name `encoding` and default value `"utf-8"`, for example, yields the text `encoding="utf-8"`. The containing parameter list collects the texts for its parameters and separates them by commas, etc. Docstrings are included in the output for classes and functions, too.

The core functionality is, thus, provided by the description of AST nodes that represent the adapters. To support traversal for code generation, we need to have links from parents to children. After moving a node, these parent-child links must be updated, which also requires that a node knows its unique parent for efficient implementation. These bidirectional links must be kept synchronized for any kind of AST node.

Moreover, we must model cross-references, which link arbitrary nodes. For example, the program text that is generated for a reference is the name of the referenced declaration. This must work even if the declaration is renamed.

For code reuse, we defined a set of abstract concepts and made them available in a small utility library[25]:

- `ModelNode` transparently manages parent links and cross-references *to* a node. Tree traversals are defined in a generic manner based on these parent and child links. Nodes must either directly or transitively inherit from this class to use the following features.
- `ContainmentReference` points to a single child.
- `ContainmentList` points to many children.
- `CrossReference` points to an arbitrary other node.

Listing 7 shows the modeling of a member access node (written as `receiver.member` in Python) using these concepts.

The children `receiver` and `member` are passed as constructor parameters (lines 2 & 3). They are then wrapped into `ContainmentReference` instances and corresponding fields are created that delegate to these instances (lines 5 & 6, `by` keyword). Delegation means that any reading or writing access to those fields gets handled uniformly by `ContainmentReference` [61]. We use this feature to ensure that assignments to child references like `receiver` also automatically update parent references in the old

[22] https://kotlinlang.org/
[23] https://ktor.io/
[24] Whitespace has a meaning in Python, so even formatting matters.
[25] https://github.com/lars-reimann/modeling

```
1 class PythonMemberAccess(
2     receiver: PythonExpression,
3     member: PythonReference,
4 ) : PythonExpression() {
5     var receiver by ContainmentReference(receiver)
6     var member by ContainmentReference(member)
7 }
```

Listing 7: Description of parent-child links in the AST for Python implemented in Kotlin.

and new child nodes, guaranteeing a valid AST. The class inherits from `PythonExpression` (line 4), which inherits from `ModelNode`.

With this machinery in place, most AST transformations are simple assignments to fields. The links in the AST automatically stay valid. Only the transformation *Group into parameter object* requires more scrutiny when replacing several parameters by one, so references to old parameters are replaced by the correct member accesses. For this, knowing which nodes have cross-references to some node has proven helpful, which is handled by `ModelNode`.

10 Evaluation (Adaptoring)

Our evaluation of the adaptoring approach was guided by four research questions, which will each be discussed in a following section:

- RQ1: How much do usage-based transformations affect the size of an API? (Section 10.1)
- RQ2: How good are the extracted preconditions? (Section 10.2)
- RQ3: How fast are the automated steps? (Section 10.3)
- RQ4: How usable is the API Editor GUI? (Section 10.4)

Since the only other similar system [25] provides no inference and no publicly accessible implementation, we could not compare our results. However, the authors themselves mentioned lack of usability as a major downside of their system.

10.1 Impact of Usage-Based Transformations on API Size

Getting client code. To quantify the impact of usage-based deletions and inference of parameter optionality, we first needed a large amount of client code. Since our own overarching goal is to develop a user-friendly, integrated library *for DS*, we chose to get client programs from Kaggle, a popular website that hosts machine learning competitions. Via a script that uses Kaggle's API[26], we could download 92,402 Python programs, many of which used the libraries we were interested in.

Choosing a library to analyze. We picked scikit-learn [33], [34] version 0.24.2 (the latest version back then) as the library to analyze due to its large physical size. Its public API consists of 270 classes, 1279 functions, and 4353 parameters. This means a function has about 3.4 parameters on average. The functions with the most parameters are the constructors of `MLPClassifier` and `MLPRegressor` (multi-layer perceptron classifier/regressor), which even have 24 parameters each. Thus, the removal of parameters could greatly boost the visibility of the remaining parameters.

Size reduction for threshold 1. Figure 11 shows the possible reduction of the API's physical size if only unused (Section 4.1) and useless (Section 4.2) API elements are removed with respect to the client programs from Kaggle. 41,867 client programs use scikit-learn in some form.

Removing unused API elements reduces the number of classes, functions, and parameters by 20.7%, 47.4%, and 51.3% respectively without affecting *any* of the 41,867 client programs. Removing all useless parameters would even lead to a reduction of 57.9%. The constructor of `MLPClassifier` was called 729

[26] https://github.com/lars-reimann/kaggle-downloader

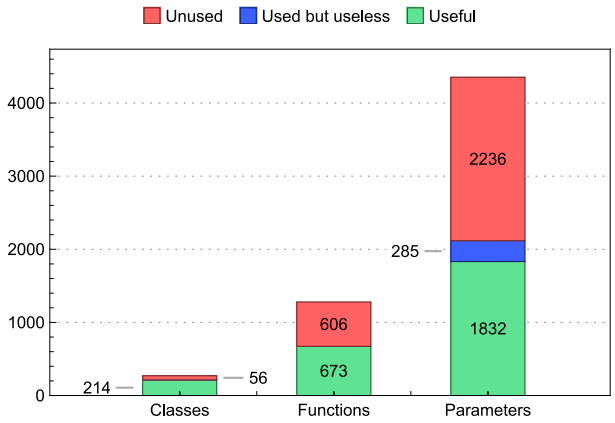
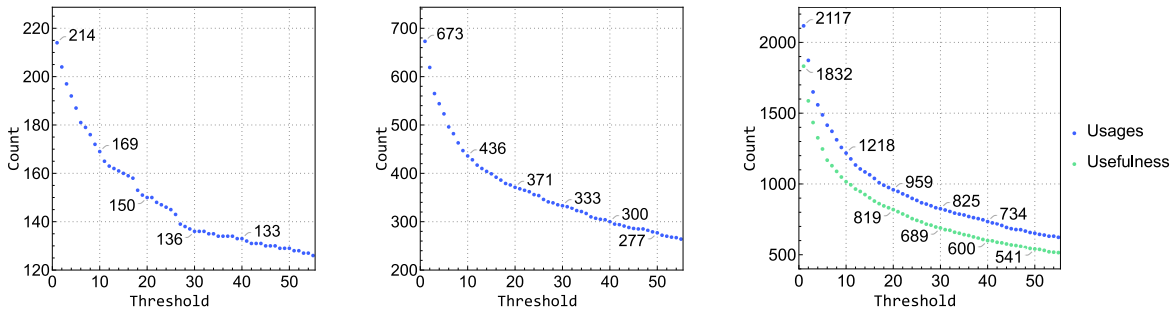


Figure 11: Number of useful, useless, and unused elements in the public API of scikit-learn. Classes and functions are useful if and only if they are used.

times and all its parameters are used. However, 5 of its parameters were always set to the same value and are, thus, useless. So, here the number of parameters could be at least reduced from 24 to 19.

Size reduction for higher thresholds. Even higher reductions in size can be achieved by using a larger threshold, as outlined in Figure 12. For `MLPClassifier`, for instance, another parameter would be removed if the usefulness threshold was set to 2, and yet another if the threshold was 3. Naturally, at some point, the loss of used functionality outweighs usability gains from simplification.

Altered parameter optionality. We also investigated the effect of the inference of parameter optionality on the conceptual size of scikit-learn. As the α we picked 0.05, a value that is frequently used for null-hypothesis significance tests. It only makes parameters optional if there is a clear most common value in the analyzed client code, opting for safety instead of simplicity. The results are shown in Table 7, with the original optionality in rows and the new optionality in columns. For parameters that were already optional, we checked whether the current default value is suitable, too.



a) Classes b) Functions c) Parameters

Figure 12: Number of public API elements of scikit-learn with higher usage count/usefulness.

Table 7: Inferred optionality of useful parameters of scikit-learn with $\alpha = 0.05$. Cases where optionality or the default value changed are highlighted.

		New API		
		Required	Optional (same default)	Optional (other default)
Original API	Required	522	N/A	2
	Optional	203	993	112

Even for this strict α, the increase in the conceptual size of the API by 201 required parameters is vastly offset by the reduction of the physical size during the previous step. Moreover, the 112 optional parameter where there is a clear most common value *other than their default value* are particularly noteworthy. Such default values are highly misleading for less experienced users.

10.2 Quality of Extracted Preconditions

We conducted two separate studies to check the correctness and completeness of the extracted preconditions. Initially, we manually reviewed the preconditions contained in the documentation of scikit-learn to create a ground truth together with seven students during the Bachelor lab "Angewandte Softwaretechnologien 2022". Then, a first extractor was created and its extracted preconditions were compared to the ground truth.

Shortcomings were then addressed in a subsequent Bachelor's thesis by Nils Vollroth [51], who refined rules to prevent several categories of false positives and added new rules to address false negatives. Afterward, we repeated our evaluation on five popular Python libraries that are used for DS. Precision was evaluated on all extracted preconditions for the entire libraries. Due to time constraints, recall was only computed for a random sample of 200 functions for each library.

We only considered an extracted precondition as a true positive (TP) if it required no changes. Extracted preconditions that had to be changed or removed are false positives (FP). False negatives (FN) are preconditions that were not extracted. The results are reported in Table 8 and Table 9.

Literal sets. Literal sets occur in all libraries and extraction generally works well, except for NumPy. Despite NumPyDoc specifying the set notation for possible values for the type section of a parameter's documentation, NumPy frequently only lists valid values in the natural language description. Additional matching rules were added for those cases, but they are less exact than the rules that match the type section.

Table 8: Precision of precondition extraction. The numbers in the table refer to the full library.

Library	Literal set			Boundary			Dependency		
	TP	FP	Precision	TP	FP	Precision	TP	FP	Precision
scikit-learn	543	45	92%	234	32	88%	224	20	92%
Matplotlib	155	18	90%	14	12	54%	1	1	50%
NumPy	95	47	67%	26	21	55%	6	11	35%
pandas	293	30	91%	10	14	42%	21	18	54%
seaborn	16	2	89%	8	0	100%	1	0	100%

Table 9: Recall of precondition extraction. The numbers in the table refer to a random sample of 200 functions of each library.

Library	Literal set			Boundary			Dependency		
	TP	FN	Recall	TP	FN	Recall	TP	FN	Recall
scikit-learn	34	2	94%	25	7	78%	21	19	53%
Matplotlib	4	1	80%	0	0	-	1	1	50%
NumPy	7	6	54%	0	1	0%	1	1	50%
pandas	14	4	78%	0	0	-	4	5	44%
seaborn	6	0	100%	4	1	80%	0	0	-

Boundary. The boundary precondition was mostly relevant for scikit-learn, where many parameters are numbers. For scikit-learn, FPs were mostly cases, where a precondition was softened by terms like "often", "typically", or "usually". FNs were caused by different wordings than expected by the matching rules. For better recall, these rules must be generalized either manually or by a machine learning model.

Dependency. Parameter dependencies were also mostly restricted to scikit-learn, where functions often have a long list of primitive parameters. This is a direct consequence of their design principle *non-proliferation of classes*, which states "Learning algorithms are the only objects to be represented using custom classes. [...] Hyperparameter names and values are represented as standard Python strings or numbers whenever possible" [34]. The reasons for FPs and FNs were identical to boundaries.

10.3 Runtime of Automated Steps

The speed of the automated steps was determined on a desktop PC with Windows 11 Pro (Build 22000), an AMD Ryzen 9 3900X processor, Samsung SSD 960 EVO 1TB, and 32 GB of main memory. Results for scikit-learn are as follows:

- The static usage analysis of 92,402 Python programs, of which 41,867 used scikit-learn, took 28 min 19s. This steps only has to be run again when new client programs are acquired. Tweaking thresholds to infer deletions or the α to infer parameter optionality can be done without rerunning this step.
- The precondition extraction from scikit-learn's docstrings took 110.3s, averaged over 10 runs. In total, the docstrings of 270 classes and 1279 functions had to be analyzed, which also contained the documentation for 4353 parameters.
- The adapter generation was done in 0.86s (mean across 10 runs). It created 55 Python files, with a total of 3,067 lines of code, and 20,683 lines of documentation. These counts were determined by cloc[27].

Overall, the automated steps are fast. After a new release of a library, sensible improvements can be inferred and implemented as adapters within two minutes if usage data is already available. Even with the usage analysis for tens of thousand client programs, a result can be obtained in half an hour. The main decelerator is the collection of client programs, which took us several hours via the rate-limited API of Kaggle. Other interesting sources, like GitHub, impose similar restrictions.

10.4 Usability of API Editor GUI

Competitor. Lastly, we compared the usability of the API Editor GUI for the transformation of an API to a competitor. Since no prototype of the only similar tool [25] is publicly available, we picked GitHub Copilot[28] (version 1.1.29.1869) in PyCharm [62] (version 222.3345.131) instead.

GitHub Copilot was only used for code-completion. Due to the regular structure of adapters, we expected inline suggestions from GitHub Copilot to be useful and accelerate programming, once the first adapter was written. The chat mode of GitHub Copilot was disabled, since participants would have been required to explain the desired API transformations in detail using natural language and the results might contain hard-to-find bugs. PyCharm was chosen due to its comprehensive Python support.

Participants. Five Bachelor students took part in the study. They were all familiar with Python, design patterns like the Adapter pattern [24], and refactoring [11]. However, they had never seen the API Editor GUI or other parts of the adaptoring approach before, nor did they know the moderator (Lars Reimann).

[27] https://github.com/AlDanial/cloc
[28] https://github.com/features/copilot

Study design. The study consisted of four main blocks. To limit fatigue, each block was constrained to 30 minutes, with 15-minute breaks between blocks 2 and 3, as well as blocks 3 and 4. In practice, all sessions were finished within two hours, including the breaks. Participants were instructed to *think aloud* [63]–[68] while working on the tasks. During the sessions, voice, face, and screen were recorded with permission.

In the first block, we explained ten general usability principles for APIs based on [26] to remind participants of good API design, without biasing them towards changes that can be done with our API Editor GUI.

1. **Visibility of system state**: Users should be able to inspect the state of the API or of a single API element, like whether a file is open.
2. **Match between system and real world**: Names and structure of the API should match users' expectations. The most well-known names should be used for the most commonly needed API elements.
3. **User control**: Users should be able to abort operations or reset the state of the API, e.g. by timing out an HTTP request after a few seconds.
4. **Consistency**: All parts of the API should have a consistent design.
5. **Recognition rather than recall**: Names should be clear and understandable, so it is obvious which API elements are needed for a task without learning the API by heart.
6. **Flexibility and efficiency of use**: Users should be able to accomplish their tasks efficiently within the limitations of the API.
7. **Minimalist design**: There should be no unneeded API elements.
8. **Help and documentation**: Should be up-to-date, understandable, and provide illustrating examples.
9. **Error prevention**: Users should be guided towards correct usage by the API, e.g. by having defaults that are correct in most cases.
10. **Error recognition, diagnosis, and recovery**: Error messages should be clear, point to the location where the error occurred, and provide instructions to resolve them.

In the second block, participants were asked to find usability flaws in the API of Matplotlib (version 3.5.3), a popular Python library for plotting, based on its documentation and the source code linked from there. This library was chosen because its domain and vocabulary can be understood intuitively, avoiding lengthy additional explanations. Participants were provided with a printout of the usability principles from block 1 as reference, but could creatively come up with their own usability flaws. Again, this prevented bias toward our tool. For each flaw, the participants wrote down a precise location in the API and a possible API transformation to resolve it.

In the third block, we briefly explained the API transformations that are supported by the API Editor to participants and provided them with a printout. They were then tasked with fixing the usability issues they found in the previous block using the API Editor GUI. Once they were done, participants filled in a standard System Usability Scale (SUS) questionnaire [69], [70] about the usability of the API Editor GUI for API transformation.

In the final block, we reminded participants of the Adapter design pattern [24] and explained its implementation in Python for global functions and instance methods of classes. Participants were again given a printout as reference. Next, participants manually implemented adapters for all API elements they wanted to change using PyCharm with GitHub Copilot. Docstrings for the implemented adapters and adapters for other functions were omitted. Lastly, participants filled in another SUS questionnaire about the usability of PyCharm and GitHub Copilot for API transformation.

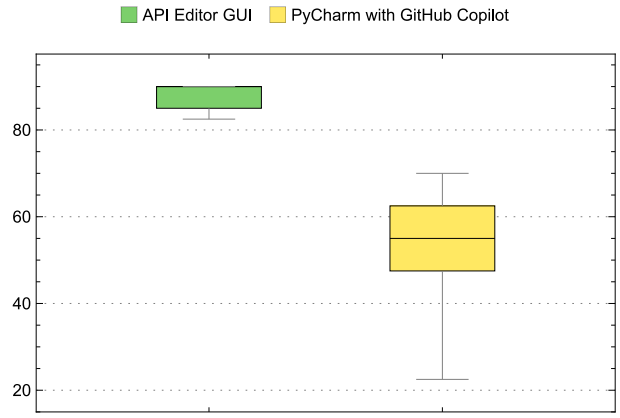


Figure 13: Comparison of the System Usability Scores for the API Editor GUI vs. PyCharm with GitHub Copilot for the creation of adapters.

Result: SUS scores. The API Editor GUI received SUS scores between 82.5 and 90, with a mean of 87.5, standard deviation[29] of 3.5, and median of 90. Mean scores above 85.5 indicate *excellent* usability on an adjective scale that highly correlates to SUS scores [72], [73].

The SUS scores for PyCharm with GitHub Copilot varied between 22.5 and 70, with a mean of 51.5, standard deviation of 18.3, and median of 55. The participant that gave the 22.5 score reported having no prior experience with PyCharm, though. Lack of experience can lead to reduced SUS scores by up to 16%[30] [74]. Nevertheless, even the highest score of 70 is still only in the *OK* range on the adjective scale [72], [73].

Thus, we conclude that the API Editor GUI provides a significantly improved usability for this task compared to PyCharm with GitHub Copilot. The SUS scores are summarized as a box plot in Figure 13.

Result: Correctness of transformations. We also counted how many API transformations to fix usability flaws of Matplotlib were handled correctly and investigated which mistakes were made frequently: All renames (41), removals (12), and additions of checks for a boundary precondition (2) were done correctly in the API Editor GUI.

Participants also wanted to combine two functions (3) and split a function in two (3), but these changes could not be done in the API Editor GUI, since it does not offer these API transformations. Implementing a generic dialog for such changes would also likely offer no usability improvement over working directly with the code because functions could be combined or split in arbitrary ways.

No participant managed to implement all adapters correctly in PyCharm. Out of 41 attempts to implement adapters to rename a function, 14 were wrong. Common mistakes included[31]:

- **Incorrect parameter list of an adapter** (9): Participants were particularly troubled by Python's complicated ordering rules for parameters caused by their optionality and other restrictions, like whether they can only be passed by position [75] or by keyword (name) [76] in calls of the function.
- **Passing a keyword-only argument by position** (9): Similarly, in calls of the original function, participants passed some keyword-only arguments by position instead.
- **Passing the original default value** (3): Normally, the value of a parameter of an adapter should be passed to the corresponding parameter of the original function. In some cases, participants copied the header of the adapter into its body, though, as a starting point for the call. The syntax for setting

[29] We applied Bessel's correction [71] (divide by the number of samples *minus one* instead of the number of samples) to all standard deviations reported in this thesis to get a more accurate estimate of the population standard deviation.

[30] Note that no participant had prior experience with the API Editor GUI.

[31] Some adapters were affected by multiple mistakes.

the default value of a parameter is identical to the syntax for passing an argument by keyword (`param="value"`), so if the participant forgot to change it, the default value was always used instead. Since this is legal syntax and semantics, PyCharm did not show an error.

More experienced Python programmers might avoid those mistakes. However, the evaluation still stresses that the GUI-based transformation is valuable to prevent mistakes related to uncommon language features, especially for novices.

10.5 Threats to Validity

Applicability to other Python libraries. The results regarding the usability of the GUI (Section 10.4) can be generalized to other Python libraries, since it is library-agnostic. Likewise, we expect the speed of the automated steps (Section 10.3) to suffice because the system performed well on the large API of scikit-learn. The quality of extracted preconditions (Section 10.2) depends on the uniformity of the docstrings of the library, though. In particular, one of the four supported docstring formats must be used. Usage analysis requires large amounts of client code, so the library must be popular. The impact of the usage analysis (Section 10.1) fully depends on the current minimalism of the library.

Applicability to other languages. Lack of static type information, the existence of dynamic imports, and conflicting formats for in-code documentation make the inference of transformations for Python difficult. Thus, we expect that the inference can also be implemented for other languages and yield at least similar results, if not better. The concepts present in the API Editor GUI and the adapter generator can be applied to other languages, too.

Participants of usability study. Only bachelor students participated in the usability study of the API Editor GUI (Section 10.4). They made several mistakes when implementing the adapters that would likely be avoided by more experienced Python programmers. Nonetheless, bugs are bound to happen when creating adapters for a large API. Note that in a real scenario, adapters for unchanged functions and docstrings would need to be added, as well, which was not emulated in the study. Moreover, training developers or employing more senior ones is costly. Finally, the creation of thousands of adapters by hand is just boring, decreasing developer happiness.

Design of usability study. Because of time constraints, participants could only detect usability flaws using the documentation and the linked source code. They never worked directly with Matplotlib in the study. Hence, they might have only found simple usability issues and overlooked other more substantial ones, which the API Editor GUI does not support yet. In any case, the API Editor GUI is not meant to be comprehensive yet. If an API transformation can be described generically and is needed often, it can be added to the system.

Finally, the length of the study might have caused fatigue when creating the adapters with PyCharm and GitHub Copilot, despite the time constraints and breaks between blocks. This might have negatively impacted the SUS scores. Nevertheless, the gap between the SUS scores of our tool and PyCharm with GitHub Copilot is huge and unlikely to vanish completely.

11 Related Work (Adaptoring)

API complexity. Deleting unused and useless API elements follows the principle of "aesthetic and minimalist design" [26] for API usability. The empirical study [23] implied that having many API elements inside the same namespace reduces the success rate of finding the required ones for a given task. Stylos and Myers found that this effect could be mitigated by proper naming and particularly distinct name prefixes to use code-completion fully [26]. Unnecessary API complexity was also the second most common issue identified in the Contextual Interaction Framework during a heuristic evaluation [77]. Lämmel et al. [78] analyzed usage patterns of Java APIs that are useful for API migration.

Several guidelines exist regarding the number of parameters a function should have: Martin [79, p. 288] suggests avoiding more than three parameters. McConnell's guideline [80, p. 178] is to use at most seven parameters, based on psychological research. Stylos and Clarke [81] even found parameterless constructors with subsequent setters to offer better usability than constructors with parameters. Complexity is also a recurring theme in usability metrics for APIs [23], [82], [83].

Debloating [84], [85] offers a way to produce a reduced library that still works correctly for *existing clients* on a given set of inputs. Adaptoring, meanwhile, creates a new, reduced API, designed to improve learnability and reduce the risk of misuse by *new clients*. It does not reduce the memory footprint of the program but increases it, adding the adapter code to the code of the original library, which must still be installed.

System configuration. Setting parameters of an API is similar to configuring a system. We, hence, also looked into literature about this topic: Yin et al. [86] found that most system failures were caused by misconfiguration. Therefore, Xu et al. [30] suggest hiding or removing rarely used configuration settings. They also found that a significant amount of system misconfigurations were caused by users incorrectly keeping default values. Ramachandran et al. [87] describe an approach on how sensible configuration settings of an enterprise system can be inferred from existing deployments. Liao et al. [88] developed a method to extract configuration constraints for automated checking. Finally, Zhang and Ernst [89] implemented a tool to fix configuration errors after software updates.

API usability. General guidelines for high-quality code like [11], [79], [80] apply especially to public APIs. Additionally, Bloch [90] as well as Myers and Stylos [26] provide specific guidance for API design. Many more studies exist about this topic: Hou and Li [91] investigated discussions from the Java Swing forum to find common learning hurdles. Grill et al. [92] evaluated the usability of an API using various research methods from human computer interaction. Scheller and Kuhn [27] compared two API designs in a study to determine their usability when instantiating classes or calling methods. Murphy et al. [93] aimed to better understand the process behind API design through developer interviews.

Code smells. Given the similarity of adaptoring and refactoring, inference of API transformations is akin to the detection of code smells [11], which indicate places where refactorings should be applied. For example, a field with a *Mysterious Name* is an opportunity to apply the *Rename Field* refactoring.

Lacerda et al. [94] reviewed literature about the definition of code smells, means to detect them, and their correspondence to refactoring. They also found that code smells and refactorings are strongly linked to code quality dimensions such as understandability. Santos et al. [95], however, concluded that no strong correlation between code smells and code quality exists, and that developers rarely agree with detected code smells. Yamashita and Moonen [96] found that developers with higher knowledge about code smells also found them more important. Fontana et al. [97], Al Dallal [98], Rasool and Arshad [99], and Di Nucci et al. [100] surveyed approaches to detect code smells. This field of research is too wide to fully do it justice here.

12 Limitations and Future Work (Adaptoring)

Inexact static analysis of Python code. As discussed previously, static analysis of Python code is difficult, due to dynamic imports and missing static types. This may cause us to underestimate usage counts and value counts. To improve results, client code can be analyzed dynamically, at the cost of vastly increased runtime and the need for sandboxing. A possible implementation would (1) instrument the library code, so all functions keep track of how often they are called and with which arguments, and (2) run all client programs against this modified library. The instrumentation can be done with a `NodeTransformer`[32], for example.

[32] https://docs.python.org/3/library/ast.html#ast.NodeTransformer

API simplification based on bad usage data. The automated inference of deletions (Section 4.1 and Section 4.2) and parameter optionality (Section 4.3) need representative client code. For a new or unpopular library, there may simply be too few client programs in total, making usage analysis pointless.

However, even for old, popular libraries, an API designer must be cautious during data collection: First, the collected client programs may require cleaning to remove invalid or closely related programs, e.g. because of repository forks. Second, for open-source libraries, it is generally impossible to collect *all* client programs. Hence, even if the usage analysis was perfect, the absence of usages in the analyzed client code does not imply that an API element is never used. In fact, as Hyrum's Law [29] implies, all API element will be used if there are enough clients. Because of this, removing only unused or useless API elements does not have an inherent benefit over picking a higher threshold. The API designer should instead choose the threshold based on their desired simplicity and expressiveness of the API.

Moreover, our usage analysis does not yet consider *when* an API element was added to a library. A recent addition is likely to have fewer usages than an API element that has been included in the library for years. Augmenting our definitions for usage counts and usefulness to consider the age of an API element could be an interesting future topic.

We can similarly discuss the other inferred transformations. The main point is that the inferred transformations are merely meant to accelerate the redesign progress. In the end, the API designer is responsible for reviewing them and assessing whether they noticeably improve the usability of the resulting library.

Missing API transformations. Our proof-of-concept implementation of the adaptoring approach only includes a small selection of possible API transformation. For example, parameters cannot be reordered yet. Such missing API transformations can be implemented later. However, as is the case for refactorings, a transformation must be needed often enough to be worth the effort to automate it. There will always be cases, e.g. when combining several libraries that require glue code to work together, where the resulting adapter code must be adjusted manually.

No evolution of manual adapter modifications. Unfortunately, the adapter generator does not know whether an adapter was later changed manually. Because of this, it would ignore any manual changes and the API designer would have to port them each time the adapter generator is invoked. This can be fixed easily by specifying a partial map from API elements to their desired adapter code and providing it as input to the adapter generator. If an API element is included in the map, the associated code is returned directly, otherwise the normal process for adapter generation is used (Chapter 6).

13 Conclusion (Adaptoring)

Adaptoring is a means to provide a new API for a library, e.g. to improve learnability and general usability. The new API is implemented using the adapter pattern and calls the original library internally. Because no code is duplicated, any bug fixes or performance improvements to the original library can be consumed immediately. The original library is not changed, so adaptoring does not break clients of the original library and avoids the need to bloat the original API further by deprecations.

Many steps of the adaptoring approach are automated to make it viable even for large libraries: Deletions (Section 4.1 and Section 4.2) and parameter optionality (Section 4.3) are inferred from usage data gathered from representative client code. Several types of preconditions are extracted from the library's documentation (Chapter 5). Given a set of transformations, adapter code gets generated automatically as well (Chapter 6). Finally, the generated adapters can be updated semi-automatically if the API of the original library changes (Chapter 8). Human intervention is only needed to resolve conflicts.

Inferred transformations can be reviewed and edited in a specialized GUI (Chapter 7). An API designer can also add transformations that cannot be inferred via this GUI, including renaming and moving an API element.

We implemented a proof-of-concept of the adaptoring approach for Python (Chapter 9) without semi-automated evolution and evaluated it on popular libraries for DS. The usage analysis indicates that, despite checking 41,867 client programs mined from Kaggle, 21% of the classes and 47% of the functions of the public API of scikit-learn are unused, and 58% of the parameters are useless (Section 10.1). Thus, vast simplifications are possible without sacrificing expressiveness. Moreover, 112 parameters have strongly misleading default values.

The extraction of preconditions from documentation achieved high precision (88% to 92%) and varying recall (53% to 94%) on scikit-learn (Section 10.2). Recall can be improved by adding more text processing rules to cover missed cases. All automated steps together took less than 31 minutes on scikit-learn and 41,867 client programs (Section 10.3). The GUI for manual API editing received an excellent mean SUS score of 87.5 (Section 10.4).

All in all, API designers can use adaptoring to *quickly* and *easily* create an improved API for a library. The result can either be used as the final product or an intermediate step to gather feedback. If needed, the API can then be refined further in fast iterations by editing the transformations to apply in the GUI and regenerating adapters. Finally, more specialized changes can be applied by editing the resulting adapter code directly, which still potentially saves writing or copying thousands of lines of code and documentation.

II A Library for Data Science

This thesis part builds on ideas from a paper published previously as part of my PhD thesis:

- *Achieving Guidance in Applied Machine Learning through Software Engineering Techniques* [101]

Abstract. Data scientists implement data science (DS) pipelines using highly expressive, specialized libraries. Even for standard DS tasks, though, developers must deal with their large application programming interfaces (APIs), and learn to combine multiple libraries with differing design philosophies and levels of abstraction, since each library only focuses on one or a few stages of the DS process. The communication between the libraries requires vast amounts of *glue code* to convert objects. This can eclipse the actual application code.

In this thesis part, we present the Safe-DS API, a *simplified*, *integrated*, and *consistent* API for DS. The API supports data preparation for tables, time series, and images, visualizations that help users understand their data with minimal configuration, supervised classical machine learning models, and neural networks for supervised tasks — all without any glue code.

The API is implemented in the open-source Safe-DS Python library, which reuses the implementation of established, efficient DS libraries. For *safety*, the library checks preconditions as early as possible at runtime and provides type hints for static type checkers, like mypy.

Python is by far the most used language by data scientists, achieving 78% usage among all respondents to the 2022 Kaggle data science (DS) survey [102]. Among other reasons, this can be attributed to the existence of highly expressive[33] libraries for all stages of the DS process, including data preparation, data visualization, and machine learning.

These libraries offer solutions to recurring problems, like filtering rows of a table or building a decision tree, and allow application developers to focus on their business logic. This greatly accelerates development. Due to their expressiveness, the libraries are applicable to many scenarios. However, the expressiveness comes at a price:

- **Large APIs**: Developers interact with a library via its *application programming interface* (API). APIs of DS libraries tend to be large, and they continue to grow: The public API of scikit-learn [33], [34], a library for classical machine learning (ML), consists of 288 classes, 1369 functions, and 5887 parameters in version 1.5.2. Compared to version 0.24.2, which we investigated in Part I, the API grew by more than 1500 parameters in roughly three years. Learning a large API and finding the API elements required for a specific task can be difficult [23].
- **Independent, specialized libraries**: Moreover, learning the API of a single library is insufficient to implement a complete DS pipeline, since they only focus on one or a few stages of the DS process, as we will see in Chapter 14. This is inevitable because maintaining a DS library for one area already requires a large team. Instead, application developers must pick and learn, say, one library for data preparation, another for data visualization, and a third for machine learning.

[33] As a reminder, we use *expressiveness* to denote the ability to express a concept at all, regardless of ease-of-use.

- **Overlapping functionality**: Despite the specialization, features of DS libraries that are often used together partially overlap, since the libraries are developed independently. For example, the table transformers of scikit-learn offer a trainable alternative to some operations of pandas [103], a dedicated library for preparing tabular data. This unnecessarily leads to a further increased combined API size.
- **Differing design philosophies**: Since the libraries are developed independently, their design principles also differ vastly. For instance, pandas makes heavy use of the indexed access operator, e.g. `table["name"]` to access a column, while scikit-learn instead uses named API elements. Thus, knowledge gained about the design of one DS library does not transfer to others, which slows learning.
- **Differing levels of abstraction**: Some libraries achieve their expressiveness by exclusively offering low-level concepts, like tensors. While these concepts are generally applicable, users are responsible for converting their data to the required form and later back to a human-readable representation. Thus, *glue code* is needed at the boundary between libraries and between a library and the user, which increases the cost of initial development and maintenance. Frequently, glue code is longer than actual application code [104], severely limiting the benefit of using libraries.
- **Lacking safety**: Low levels of abstraction also lead to worse error messages. Instead of alerting a user that a given table is lacking the column `age`, the library may show an error that the shape of a tensor is wrong. This slows debugging, which again increases cost.

Goal: A usable API for DS. Our goal in this thesis part is to design a new DS API, the *Safe-DS API*, that can be used to easily and quickly get insights into data. The API should improve upon the issues mentioned above, at the cost of some expressiveness. Hence, it is an explicit non-goal to provide an API for DS researchers to implement new algorithms, or DS practitioners who need access to all hyperparameters of models to maximize their quality. These areas are covered well by existing libraries.

The API should initially support data preparation for tables, time series, and images. Moreover, it should offer visualizations that help users understand their data with minimal configuration. Finally, it should include supervised classical ML and supervised neural networks (NNs). We picked these areas, so the API alone is sufficient for real-world applications. In particular, many of the programs we downloaded from Kaggle (Section 10.1) could be replicated with the Safe-DS API.

Part outline. The remaining content of this thesis part is structured as followed:

- Chapter 14 surveys the state of the art regarding existing libraries for the DS areas mentioned above.
- Chapter 15 details the goals we want to achieve with the Safe-DS API and its implementation.
- Chapter 16 describes the design of the Safe-DS API that takes the API of existing libraries into account but improves them regarding the goals outlined in Chapter 15.
- Chapter 17 discusses our choice of existing libraries to implement the API using adaptoring (Part I), and other aspects regarding the implementation of the API as a new Python library.
- Chapter 18 contains the evaluation of the library regarding usability, learnability, and development time.

14 State of the Art: Python Libraries for DS

DS libraries exist for many programming languages, but as Python is essentially the lingua franca of data scientists [102], we restrict our overview of the state of the art to Python libraries. Covering libraries for other languages would be impractical due to space requirements, and we refer to dedicated survey papers [105], [106] instead.

Table 10: General information about popular Python libraries for DS from October 28, 2024. The column "Regular use" indicates the percentage of respondents to [102] that regularly use a library. The data for stars, contributors, and downloads only includes the main repository and package.

Category	Library	Regular Use (% Respondents)	GitHub Stars	GitHub Contributors	PyPI Downloads in Last Month
Table preparation (Section 14.1)	pandas	Not asked	43,627	3,335	258,740,086
	Polars	Not asked	30,004	505	8,545,714
Time series preparation (Section 14.2)	pandas	Not asked	43,627	3,335	258,740,086
	Polars	Not asked	30,004	505	8,545,714
Image preparation (Section 14.3)	Pillow	Not asked	12,221	415	120,940,191
	Pillow-SIMD	Not asked	2,179	No data	34,925
	scikit-image	Not asked	6,083	586	13,581,151
	TorchVision	Not asked	16,157	617	13,594,830
	TensorFlow	33.1%	186,223	3,607	17,920,583
Data visualization (Section 14.4)	Matplotlib	58.4%	20,198	1,477	75,912,884
	seaborn	43.8%	12,516	200	17,876,343
	plotly	21,2%	16,190	255	18,847,355
	Bokeh	3.2%	19,317	610	3,895,577
	Vega-Altair	1.3%	9,306	157	21,580,192
Classical ML (Section 14.5)	scikit-learn	47.5%	59,906	2,938	77,150,989
	XGBoost	18.7%	26,236	625	27,740,518
	LightGBM	8.1%	16,653	319	8,954,094
	CatBoost	4.9%	8,067	377	3,203,485
Neural networks (Section 14.6)	TensorFlow	33.1%	186,223	3,607	17,920,583
	Keras	27.4%	61,939	1,266	13,258,430
	PyTorch	21.6%	83,381	3,554	32,411,038
	Hugging Face	5.6%	134,088	2,811	43,366,154
	JAX	1.1%	30,345	703	3,869,147

Even the Python Package Index (PyPI)[34] alone, however, hosts thousands of Python libraries related to data science. Thus, we only include libraries that are relevant to the areas mentioned in the introduction, and further filter libraries based on their popularity according to the 2022 Kaggle survey [102], stars and number of contributors on GitHub, and downloads from PyPI[35].

Neither of these metrics is perfect: Kaggle lets users run programs on their servers, which provide a pre-installed set of DS libraries. Users can install additional ones, but are still more likely to regularly use the pre-installed ones. The number of GitHub stars is naturally higher for older libraries and can be influenced easily, e.g. if a large company asks all their employees to star a repository. The number of contributors does not consider the number and size of their contributions. PyPI downloads also include cases, where a library is downloaded as a *transitive* dependency, but omit cases where a library is downloaded from other sources, like conda-forge[36].

[34]https://pypi.org/

[35]The data can be retrieved from https://pypistats.org/.

[36]https://conda-forge.org/

Nevertheless, we lack an alternative, and when combined, the metrics provide a useful estimate of the current popularity and health of a project. Based on these metrics, we selected the libraries shown in Table 10 for further investigation. We have since updated the table with data from October 28, 2024. In the following sections we will look into each library in depth, including the functionality if offers, its API design, and other criteria like efficiency.

14.1 Data Preparation for Tables

pandas. pandas [103] is the most popular library for table preparation. It offers two core data containers:

- `DataFrame` is a two-dimensional structure, which can represent tables. Besides the data, it includes row labels (also called *index*) and column labels (also just called *columns*).
- `Series` is a one-dimensional structure with an index. Typically, `Series` is used to represent a column. There are, however, also cases where `Series` represents a row, e.g. when iterating over the rows of a `DataFrame`. In this case, the index represents *column* labels, rather than the usual *row* labels. As `Series` is also used for rows, the data contained in them need not be homogeneous.

Operations are typically implemented as methods on the data containers. Compared to global functions, this approach offers better code-completion because suggestions are filtered by the type of the call receiver. To further guide code-completion, some methods are additionally grouped into namespaces: `Series.dt`, for instance, contains methods for working with temporal data, while `Series.str` contains methods for text.

Many `DataFrame` methods take a parameter `axis`, which determines whether the operation should loop over rows (value `0`/`"index"`) or columns (value `1`/`"columns"`). For example, the method `DataFrame.dropna` either removes rows or columns with missing values. Other operations are hard-coded to apply to either rows or columns. The `axis` parameter also appears on some methods of `Series`, even though it is meaningless on a one-dimensional data structure.

Moreover, some common tasks have corresponding operators. In particular, the indexed access operator `[]` appears frequently in code that uses pandas. It can be used to extract a single row or column of a `DataFrame`, a scalar of a `Series`, projection (picking several columns of a `DataFrame`), and filtering (picking several rows of a `DataFrame`) — among other tasks.

For data visualization, pandas offers a thin wrapper around Matplotlib [107] (Section 14.4) that can create various plots for a `DataFrame`, like histograms or scatter plots. All visualization methods are grouped under the `plot` namespace of `DataFrame`. Users can opt to use another plotting library as backend, as long as it provides the expected interface. In any case, pandas fully exposes the API of the used plotting library, which must be used to customize the resulting plots.

Polars. The first release of Polars [108] happened in March 2021. Since then, the library has quickly gained popularity, presumably due to its focus on performance. According to Polars own benchmark[37], it is between 10 to 100 times faster than pandas, if reading the data from disk is excluded, where overall performance is dictated by I/O speed. This also matches our experience in practice. The performance gain can be attributed to a number of factors:

- **Rust backend**: While Polars offers a seamless Python API, the core of the library is implemented in Rust[38]. This programming language focuses on performance and offers better code optimization compared to Python. Hence, the library code itself is faster.
- **Lazy evaluation and query optimization**: Where possible, operations are not executed immediately, but deferred until their results are needed. This allows powerful query optimization. For

[37] https://github.com/pola-rs/polars-benchmark
[38] https://www.rust-lang.org/

example, if the user first doubles all numbers in the table and then prints the first three rows, it suffices to transform only the first three rows. Polars even applies this principle to some input functions: `scan_csv` lazily reads a CSV file, so only the data required by later operations is read from disk, saving both time and memory.
- **Multi-threading**: Work is automatically divided between CPU cores. This applies to individual operations, and also operations that are independent according to the execution plan derived by the query optimizer.
- **Vectorized operations**: Data is stored in a columnar format. This allows processing homogeneous data in a vectorized manner (single instruction, multiple data, SIMD). Polars is also currently working on GPU support for further performance gains.

At first glance, the API of Polars appears to be similar to pandas. There are classes `DataFrame` and `Series` for two-dimensional and one-dimensional data, respectively. However, a closer look reveals that the API is entirely different:

- `DataFrame` encapsulates two-dimensional data and its schema, i.e. a map from column names to types. The schema can be provided explicitly or inferred from the data. There are no row labels (index). All methods of `DataFrame` are executed *eagerly*, so query optimization is not possible.
- `LazyFrame` also contains two-dimensional data and its schema. But it implements a subset of the methods of `DataFrame` in a lazy manner and should be used instead of `DataFrame` whenever possible. Essentially, it uses the contained data as a starting point and remembers the operations applied to it. To view the resulting data, clients must convert the `LazyFrame` to a `DataFrame`, at which point an optimized execution plan is computed and run. Viewing the resulting schema is typically a fast operation and does not require running the full execution plan. We will come back to this property in Section 25.5.
- `Series` has homogeneous, one-dimensional data, a name, and a type. It is only used for columns.
- `Expr` (expression) is essentially the lazy counterpart to `Series` and largely mirrors its API. It describes operations on a column without running them immediately. For example, the expression `pl.col("A").mul(2)` first selects the column with the name `"A"` and then multiplies its values by `2`. Many methods of `DataFrame` and `LazyFrame` expect expressions as arguments, like `with_columns` which adds columns to a `DataFrame` or `LazyFrame`. The values of the new columns are described with expressions.
- Rows do not have a specific container, but are instead a plain dictionary from column names to values in the few instances where they appear. This representation is highly inefficient in terms of speed (no vectorization) and memory, and should be avoided if possible.

Similar to pandas, most operations are methods on data containers and namespaces are used to group related operations together: `Expr.dt` contains temporal operations, and `Expr.str` contains text operations, for instance.

Methods of `DataFrame`/`LazyFrame` either work only on rows or columns. There is no `axis` parameter to switch modes. The `[]` operator is rarely used and typically replaced by expressions.

Polars offers a thin wrapper around Vega-Altair [109] for data visualization (Section 14.4). Visualization is implemented by methods under the `plot` namespace of `DataFrame`, for plots that need more than one column, and `Series`, for plots that only need a single column. All methods return instances of classes from Vega-Altair. Users must interact with these instances to customize plots.

14.2 Data Preparation for Time Series

Preparing time series does not require additional, dedicated libraries. Instead, both pandas and Polars can work with arbitrary temporal data contained in a table.

pandas. pandas [103] offers two main data types to represent temporal data: A `Timestamp` is a single point in time. A `Period` has a start time and duration, e.g. one day. Its end time is computed from the start time and duration. Both types can be used as row labels (index).

Moreover, pandas has two data types for temporal arithmetic: A `Timedelta` is an absolute duration, like 24 hours. A `DateOffset` is similar, but also considers special calendar events, like leap years or daylight savings time. A one day `DateOffset` may, thus, correspond to a 23, 24 or 25 hour `Timedelta`. Both can be added to or subtracted from a `Timestamp` to compute a new `Timestamp`. Subtracting two `Timestamp`s yields a `Timedelta`.

Additional temporal operations are available as methods of `Series` (Section 14.1) grouped under the `dt` namespace. This includes extraction of time components like minutes or seconds, and calendar operations like whether a year is a leap year.

Finally, pandas supports windowing, i.e. applying some operation to neighboring rows. The size of the window is generally based on the number of rows of consider. However, for temporal data, the window size can also be a duration, e.g. using all values within one day as input.

Polars. Polars has three main temporal data types[39]: `Date`, `Time`, and `Datetime` represent a date (without a time), time (without a date), and a date with a time respectively. There is no dedicated data type for periods, which must instead be emulated using two columns, one for the start time and another for the end time.

`Duration` is the only additional data type for temporal arithmetic. It represents an absolute duration, so one day is always 24 hours, identical to pandas's `Timedelta`. A `Duration` can be added to a `Date` or `Datetime` to produce a new `Date` or `Datetime`, and it is the result when subtracting two `Date`s or `Datetime`s. Calendar arithmetic is instead implemented by a method `offset_by` in the `dt` namespace.

This `dt` namespace groups all temporal operations together and is available for `Series` and `Expr` (Section 14.1). The provided functionality is similar to pandas: There are methods to extract date components, and calendar operations, like a leap year check.

Like pandas, Polars also supports windowed operations on temporal columns. The most flexible approach is to first group data by a temporal column and then aggregate the values in each group.

14.3 Data Preparation for Images

Pillow. The core of Pillow [110] is the `Image` class that represents a single image. An image can be manipulated by calling methods on an `Image` object. Some operations, like blurring or sharpening do not have dedicated methods, but are instead implemented as instances of `ImageFilter`, which can be applied generically via the `filter` method of `Image`.

Regarding efficiency, Pillow solely runs on the CPU. It also has no special support for batch operations, i.e. applying the same transformation to multiple images, either. This must instead be implemented by manually looping over the images and transforming each one individually.

Pillow-SIMD. A fork of Pillow, called Pillow-SIMD [111], provides the same API as Pillow, but uses SIMD instructions of x86 processors to greatly accelerate some operations, like resizing or blurring an image. Because of this, the library is only compatible with x86 processors. On a benchmark provided by Pillow, this lead to an acceleration by a factor of 4-6 compared to the original Pillow.

scikit-image. The API of scikit-image [112] is drastically different from Pillow. It has no own class for images, and instead represents them as low-level NumPy [113] arrays. Because of this, its API solely consists of global functions.

[39] https://docs.pola.rs/user-guide/transformations/time-series/parsing/

Like Pillow, scikit-image only runs on the CPU. While scikit-image has a container `ImageCollection` to hold multiple images, its global functions always expect a single NumPy array as input, i.e. one image. Hence, batching requires manual looping. scikit-image depends on Pillow, so its installation size is always larger than using Pillow directly.

TorchVision. TorchVision [114] is part of the PyTorch ecosystem [115], which we will cover in Section 14.6. Images are represented as PyTorch tensors. There is also a class `Image`, but it is a simple subclass of PyTorch's `Tensor` class, with little extra functionality.

Image operations are implemented as global functions. Additionally, TorchVision defines special neural network layers for *data augmentation*, i.e. artificially increasing the size of the dataset before feeding it to an ML model [116]. For example, `RandomRotation` rotates an image by a *random* amount.

TorchVision has GPU support and all image operations can be applied to a batch of images. Moreover, since images are already represented as PyTorch `Tensor`s, they can be used in neural networks implemented with PyTorch without further conversion.

TensorFlow. TensorFlow [117], which we discuss in Section 14.6, has a built-in equivalent to TorchVision, where images are represented by TensorFlow tensors. Image operations are provided by global functions. Similar to PyTorch, there are also related layers for neural networks, like the `RandomRotation` layer to rotate images by a random amount.

TensorFlow has GPU support and can transform a batch of images at once. As images are represented by TensorFlow tensors, no conversion is required to use them as input to a neural network implemented with TensorFlow.

14.4 Data Visualization

Matplotlib. Matplotlib [107] is a library to create highly customizable visualizations. It has generic support for interactivity, like zooming or panning a visualization, or displaying the coordinates of the mouse cursor. The library mostly focuses on static visualizations, though.

Matplotlib supports a wide range of plots that can visualize pairwise data (e.g. scatter plots), distributions (e.g. box plots), gridded data (e.g. contour plots), or three-dimensional data (e.g. three-dimensional scatter plots).

The basic structure of a visualization is described by composing objects (Composite design pattern [24]): `Figure` objects stand for a single visualization. A `Figure` can contain multiple sub-visualization, e.g. arranged in a grid. Each sub-visualization is represented by an `Axes` object. An `Axes` contains `Artist` objects, which handle the actual drawing.

There are two different APIs offered by Matplotlib: In the *explicit API*, users directly instantiate and interact with `Figure` and `Axes` objects. They then call methods on these objects to set titles, configure colors, etc. `Artist`s are typically not handled explicitly, but instantiated and added in one step by calling methods on an `Axes`.

In the *implicit API*, Matplotlib hides `Figure` and `Axes` objects from users. They instead call global functions and Matplotlib tries to guess when new `Figure` and `Axes` objects should be created, how they should be composed, and which one should be manipulated, e.g. when setting a title. Use of the implicit interface is discouraged by Matplotlib these days [118], but it is not deprecated.

seaborn. Another library for static visualizations is seaborn [119]. It is based on Matplotlib and provides several additional plot types, like heatmaps[40], and a consistent default look for a subset of the plots for pairwise data and distributions offered by Matplotlib.

[40] Heatmaps can also be created with Matplotlib primitives, but replicating seaborn's result requires some effort.

The API of seaborn consists of global functions that create Matplotlib `Artist`s and, by default, add them to the current `Figure` and `Axes` managed by the implicit API of Matplotlib. For functions that only create a single plot, users can pass the `Axes` object to which the plot should be added as a parameter.

The visualization can be customized partially via parameters of the plot functions. To configure an `Axes` or `Figure` object, like when adding titles, users must use the API of Matplotlib directly, however.

plotly. plotly [120] is a library for interactive plotting. It largely supports the plot types of Matplotlib and adds dedicated plot types for geographical data. The library offers the same generic interactivity as Matplotlib, including zooming and panning. Moreover, elements of plots can react independently to events, like mouse clicks, dragging, or hover. For example, clicking on an entry in the legend of a plot may toggle the visibility of the associated entries.

A visualization is represented by a `Figure` object. A `Figure` can compose other objects, like a `Layout` to describe subplots, and subclasses of `Trace` for specific plots. Users can directly instantiate and compose those objects. This low-level API gives them full control over the resulting plot.

There is also a high-level API consisting of global functions, like `scatter`, that handle the composition internally and return a complete figure with useful default interactivity. Users can configure the result via parameters of those global functions.

Bokeh. Bokeh [121] is another library for interactive plotting. The set of provided plots is comparable to plotly. Interactivity is provided by so-called *tools*. These cover generic interactivity (e.g. zooming and panning), inspection (e.g. tooltips), and editing. Additionally, *widgets* can be added to plots, which allow users to enter data that can be consumed by plots.

A visualization is represented by a `Plot` object. Tools, layouts (e.g. a grid for subplots), and other elements are then added to the object and create the final result via the Composite design pattern [24]. A `Plot` can be configured either by setting parameters upon instantiation or via subsequent method calls and assignments to attributes.

The API also has a `figure` class [*sic*], which is a subclass of `Plot` that has additional methods to create plots and add them to the `figure` in one step. Parameters of the methods of `figure` are used to configure each plot. Interactivity must always be configured explicitly, even when using the high-level `figure`-API.

Vega-Altair. Vega-Altair [109] can create interactive plots, too. Instead of offering a predefined set of plots, the library provides graphical primitives, called *marks*, that react to data. For example, a circle may get its x-coordinate from column `A`, its y-coordinate from column `B`, and its size from column `C`. Using those primitives, various plot types can be created, such as a scatter plot using the circles.

Interactivity is based on so-called *parameters*. These can be set by *widgets*, which are input elements like sliders or text boxes, or events like mouse clicks. They can then be consumed by plots to, say, pan them or select certain data points.

The `Chart` class forms the core of the API. It provides methods to add marks to the chart and bind their properties to the data, as described above. By default, plots are static, and any interactivity must be added explicitly using interactivity parameters.

14.5 Classical Machine Learning

scikit-learn. scikit-learn [33], [34] provides a wide array of classical ML algorithms (i.e. no deep learning) for supervised tasks (regression/classification) and unsupervised tasks, such as clustering. It can also prepare data for use with these ML algorithms, e.g. by transforming categorical data into numerical data or filling missing values (*imputation*). Additionally, scikit-learn implements metrics to gauge the results of ML models and offers methods to find suitable hyperparameters with little code.

According to [34], scikit-learn's API follows five core principles:

- **Consistency**: All objects are based on a few, small interfaces. An `Estimator` can be trained by calling its `fit` method. A `Predictor` is an `Estimator` that can also create predictions via its `predict` method. All ML models are `Predictor`s. A `Transformer` is an `Estimator` that can manipulate data with its `transform` method, for example for imputation.
- **Inspection**: Hyperparameters and learned parameters can be retrieved via attributes.
- **Non-proliferation of classes**: Classes are only created for `Estimator`s. Data is represented by NumPy [113] arrays or SciPy [122] sparse matrices. Hyperparameters are primitive Python types, like strings.
- **Composition**: The API focuses on providing core building blocks that can be composed [24] to create new operations. For example, scikit-learn implements a general `VotingClassifier` that can be parametrized by arbitrary other classifiers. The `VotingClassifier` can then be trained like any other model, which transparently trains the internal models. It classifies data by letting the internal models vote on the answer.
- **Sensible defaults**: Parameters have default values, so functions provide a suitable baseline solution without additional user configuration.

The API of scikit-learn is separated into multiple packages. Most packages describe the task they solve, like `sklearn.cluster` for clustering or `sklearn.impute` for imputation. Supervised ML algorithms are instead grouped loosely by some concept that underlies them. For instance, the package `sklearn.tree` contains classification and regression models that are based on decision trees. Random forest models are instead contained in the package `sklearn.ensemble`.

Originally, scikit-learn ran on the CPU only. However, in 2023 an effort started to add GPU support for the estimators that would benefit the most [123].

XGBoost, LightGBM, CatBoost. Gradient boosting [124], [125] iteratively builds a model using *weak learners*, which are models that are barely better than random guessing (classification) or always predicting the mean value (regression). Typically, small decision trees are used as weak learners. In each iteration, a new weak learner is trained to fix the errors of the current model and then integrated to create a new intermediate model.

Gradient boosting frequently produces good results on tabular data, which explains the popularity of the specialized libraries XGBoost [126], LightGBM [127], and CatBoost [128] (Table 10). All libraries provide an API that is compatible with the `Predictor` interface of scikit-learn. Compared to scikit-learn's implementation of gradient boosting, they offer improved performance and GPU support.

14.6 Neural Networks

PyTorch. PyTorch [115] offers a low-level API to work directly with tensors (multidimensional arrays containing homogeneous data). It supports automatic differentiation of functions, which is used for backpropagation during model training. This low-level API can be used for arbitrary acyclic computational graphs, e.g. for new NN architectures. The graph structure can be altered during training.

Additionally, PyTorch has a high-level API to describe NNs: A `Module` may contain *parameters*, which are trainable tensors. Moreover, it has a `forward` method that gets a tensor as input and computes an output tensor, perhaps using its parameters in the computation. `Module`s can call other modules in their `forward` method (Composite design pattern [24]), so they are suitable for describing individual layers of NNs and complete NNs. Gradients are computed automatically based on the flow described in the `forward` method.

There are many predefined layer types, like a `Linear` or `Dropout` layer, implemented as subclasses of `Module`, to quickly build NNs. Finally, the `Sequential` class can be used to create an NN from a linear sequence of layers without having to subclass `Module` and implement a `forward` method.

Data is represented by creating a subclass of `Dataset`. A `Dataset` instance can be wrapped in a `DataLoader` object, which handles iteration over the data, including shuffling, batching, and parallel loading. This simplifies the training loop.

The training loop must be programmed explicitly, including the definition of a loss function and optimizer, looping over the training data, computation of the current model output and loss, as well as triggering backpropagation.

PyTorch can run on most current CPUs and supports GPU acceleration using NVIDIA's CUDA[41], AMD's ROCm[42], and Apple's Metal[43]. Since version 2.3.0, which was released in April 2024, no official binaries have been created for macOS on Intel x86 processors anymore[44].

TensorFlow. The version 2 API of TensorFlow [117] is similar to PyTorch: TensorFlow has a low-level API to work with tensors, perform automatic differentiation, and define acyclic computational graphs. Likewise, TensorFlow defines a `Module` class, that layers and complete NNs can inherit from. A `Module` can contain *variables*, which are tensors that can be trainable or untrainable. The forward computation must be implemented in a `__call__` method, which gets a tensor as input and returns a tensor. `Module`s can compose other `Module`s.

TensorFlow also includes the Keras API (see dedicated section below), which provides predefined layers, a `Sequential` class to quickly define NNs based on a sequence of layers, and a subclass of `Module` called `Model`, which can be trained without manually programming a training loop. If the basic `Module` API is used directly, training loops must be programmed explicitly using steps similar to PyTorch, only with differently named API elements.

Data gets represented by an instance of the `Dataset` class, which can be constructed from tensors. TensorFlow also has a few convenience methods to create a `Dataset` object, e.g. from a CSV file. A `Dataset` can be batched and shuffled by calling methods on the object.

TensorFlow can run on the CPU and officially supports CUDA for GPU acceleration. There are also third-party builds for ROCm and Metal, but they are lagging the official releases by several months.

JAX + Flax. JAX [129] is a library for tensor computations and includes tools for automatic differentiation. It provides an API that closely mirrors the API of NumPy [113], so JAX can often be used as a drop-in replacement for NumPy. Compared to NumPy, which can only run eagerly on the CPU, JAX offers just-in-time compilation to optimize code, and can run on various GPUs and TPUs (tensor processing units). However, regarding GPU/TPU acceleration, there is no direct support for Windows and only experimental support for macOS so far.

JAX itself does not have a high-level API for NNs. This is instead delegated to libraries built on top of JAX. Keras (see below) can use JAX as a backend. There are also NN libraries that specifically target JAX, like Flax [130]. The modern API of Flax, called NNX, is again very similar to PyTorch and TensorFlow: Layers and models inherit from `Module`, remember learned values in *parameter* tensors, and must implement a `__call__` method to define the forward pass. `Module`s can compose other `Module`s. Flax also has predefined layers and a helper function to create sequential models.

[41] https://developer.nvidia.com/cuda-toolkit
[42] https://www.amd.com/en/products/software/rocm.html
[43] https://developer.apple.com/metal/
[44] https://github.com/pytorch/pytorch/releases/tag/v2.3.0

Flax does not have its own abstraction for datasets. Its documentation instead guides users towards the `Dataset` class of TensorFlow. Training loops must be written explicitly, in a fashion comparable to PyTorch and TensorFlow. Flax supports the same platforms as JAX.

Keras. Unlike the previous libraries, Keras [131] does not implement tensor computations or automatic differentiation. Instead, it uses PyTorch, TensorFlow, or JAX as the backend for these tasks and provides the same high-level API regardless of the chosen backend. TensorFlow includes Keras, nowadays.

Layers inherit from the `Layer` class, contain *state* for trained weights, and implement a `call` method to describe the forward pass. A `Layer` can be composed of other `Layers`. Keras includes a large collection of predefined layers. A `Model` is composed of `Layers` and can be defined in three different ways:

- The `Sequential` class can be instantiated with a linear sequence of layers.
- In the functional API, all layers receive the previous layer as an argument, so arbitrary directed, acyclic layer graphs can be created. Afterward, an instance of `Model` is created and the input and output layers are passed as arguments.
- Finally, `Model` can be subclassed, which requires implementing a `call` method to define the forward pass. The `call` method receives an input tensor and produces an output tensor. In the end, the custom `Model` subclass must be instantiated.

Regardless of the chosen way to define the model, the user ends up with a `Model` object that can be trained consistently: First, the `compile` method must be called to configure the optimizer and loss to use during training and various other options. Then, `fit` receives the training data and handles the training internally. Users need not write an explicit training loop.

Keras can be used on any platform that the chosen backend is available on, and also inherits their capabilities for hardware acceleration.

Hugging Face. The Transformers library [132] from Hugging Face (stylized as 🤗 Transformers) focuses on pretrained models. Models for recurring tasks, like image classification or speech-to-text, are hosted in a central repository, called the Hub. These models can be implemented in PyTorch, TensorFlow, or JAX. Keras models are also compatible, since it uses those libraries as backend.

Transformers offers a unified API to fetch models and use them for inference on the user's own data. Moreover, models can be fine-tuned, i.e. retrained on the user's data. For this purpose, there is a `Trainer` class that users can instantiate with a pretrained model, and their training data. Calling the `train` method on the `Trainer` object then starts the fine-tuning.

Regarding data, Hugging Face offers another library, called Datasets (stylized as 🤗 Datasets) with two main classes: A `Dataset` contains tabular data and always gets fully loaded into memory. An `IterableDataset` only loads the portion of the data that is currently needed and is, thus, suited for data that does not fit into memory. Binary data, like images or audio, are stored as textual paths in datasets and can optionally be loaded and decoded automatically upon creation of the dataset.

Datasets can be created from local data. Additionally, the Hub hosts a collection of more than 200,000 datasets, which can be used to train or benchmark new models.

14.7 Research Gap

The review of the state of the art provides further evidence for the issues raised in the introduction:

- **Large APIs & independent, specialized libraries**: A typical stack to train a classical ML model on tabular data, consists of pandas, scikit-learn, and potentially Matplotlib. The APIs of each of these libraries encompass hundreds to thousands of API elements, so even if users know which library must be used at each step, finding the correct API element might be difficult. This issue is

worsened because some libraries duplicate large parts of their API, like Matplotlib with its implicit and explicit interface (Section 14.4).

- **Overlapping functionality**: Moreover, the libraries partially overlap, so sometimes it is not even obvious which library must be used. The `plot` namespace of pandas can create basic plots, but for any customization, Matplotlib must be accessed directly.

  Likewise, as mentioned in the introduction, some `Transformer`s of scikit-learn are closely related to operations of pandas. For example, the `StandardScaler` of scikit-learn subtracts the mean from a column and divides by its standard deviation. pandas can also compute means and standard deviations. The main difference is that a `StandardScaler` can *learn* the mean and standard deviation once from a table, so it then transforms, say, training and test data identically.

- **Differing design philosophies**: Differences in design philosophies are numerous: Names of API elements in scikit-learn rarely contain abbreviations and consistently use underscores between words. Matplotlib on the other hand liberally uses abbreviations, even for library-specific concepts, such as the `gca` method of `Figure`, which stands for "get current axes". Use of underscores is also inconsistent in Matplotlib: `Figure` has an `add_axes` method to add an `Axes` object and a `delaxes` method to delete it again.

  One more example regarding design philosophies: The `fit` method of scikit-learn `Estimator`s mutates them in place. Instead, methods of a pandas `DataFrame` return a new `DataFrame` by default[45] and keep the original one unchanged. If users apply the scikit-learn logic and forget to assign the result of the `DataFrame` method call, the call does nothing.

- **Differing levels of abstraction**: The main abstraction provided by pandas is the `DataFrame` class. However, Matplotlib and scikit-learn `Predictor`s expect NumPy arrays as input. Hence, glue code is required to turn a `DataFrame` into arrays. This step is easy and efficient, since a `DataFrame` is backed by NumPy arrays by default and a `DataFrame` has methods to access this low-level representation.

  However, the `predict` method of `Predictors` also only returns a one-dimensional NumPy array. To associate predictions with input for interpretation, users must attach this array back to the input `DataFrame` as an extra column. This already needs longer glue code, more intricate knowledge of pandas, and overall longer development time.

- **Lacking safety**: Converting a `DataFrame` to NumPy arrays loses important information, like column names. Hence, scikit-learn can no longer include column names in the errors that are raised if `predict` is called with invalid data. Instead, it can only point out discrepancies regarding the number of columns or column types. Worse, if columns of the same type are accidentally swapped in training and production data, scikit-learn silently accepts it.

  scikit-learn and Matplotlib also lack support for static safety that can be provided by linters like mypy [133]. Neither library adopted type hints [28]. Moreover, many functions of Matplotlib have a `**kwargs` parameter, that consumes *any* named argument, instead of explicitly listing the legal parameters. This even prevents checking whether a parameter exists, and breaks code-completion.

15 Goals (Library)

API goals. As mentioned in the introduction, the Safe-DS API should allow clients to easily and quickly get insights into data, at the cost of some expressiveness. In particular, we focus on the following areas (more details in Chapter 16):

- **Simplicity vs. expressiveness**: As we showed in Section 10.1, the full expressiveness of scikit-learn is not needed for many applications. This also applies to other DS libraries. So, we want

[45] For some methods this behavior can be changed by setting the `inplace` parameter to `True`.

to design a simplified DS API by removing seldom-used API elements and redundancies between DS libraries. Naturally, the expressiveness of the Safe-DS API should still suffice for real-world applications.

- **Integration & consistency**: The API should initially encompass data preparation for tables, time series, and images, data visualization, supervised classical ML, and supervised NNs. The design of the API should be consistent throughout. If users learn a principle for one part of the API, they should be able to recognize and apply it elsewhere.
- **High-level abstractions**: Where possible, users should work with high-level concepts like tables, images, or layers, instead of low-level concepts like tensors. Names of API elements should follow established DS terms. This provides a closer mapping to the DS domain model.
- **Discoverability**: Simplicity, and a direct mapping between domain concepts and API elements via high-level abstractions are both means to improve *discoverability*, i.e. the ability to easily find the API elements needed to solve a given task. As the API should be expressive enough to solve complete DS tasks, even the simplified Safe-DS API will be fairly large. Organizing API elements in a way to facilitate discoverability is crucial.
- **Safety**: The design of the API should eliminate potential sources for mistakes, where possible. The remaining preconditions should be checked as early as possible, and the program should fail with a helpful error message if they are violated. Failing early is particularly important due to the long runtime of DS programs. Moreover, the API should provide type hints [28], so users can check types statically with a tool like mypy.

Implementation goals. The implementation of the API (Chapter 17) should meet certain criteria, too. These goals are secondary to the API goals, which means we focus foremost on optimizing the API before optimizing the implementation:

- **Code reuse**: The cost to develop and maintain a fresh library for complete DS applications would be prohibitive. Moreover, it would be impossible to compete with the speed at which the existing libraries improve in terms of features, quality, and efficiency. The API should, therefore, be implemented as an adapter library that calls existing libraries using our adaptoring approach (Part I). This allows us to immediately consume bug fixes and performance improvements to the original libraries. In Chapter 14, we looked at suitable DS libraries we can use as our foundation.
- **Efficiency**: The implementation should use fast libraries as its basis. Moreover, our abstractions and additional safety checks should add minimal overhead. The number of dependencies should be minimized, to keep the download and installation size down.
- **Platform & hardware compatibility**: The implementation should run on current Linux, macOS, and Windows systems. Execution on the CPU should always be supported, as well as running tasks on the GPU that heavily benefit from it. Switching between CPU and GPU should be transparent to clients. Supporting NVIDIA CUDA is the minimum requirement regarding GPU acceleration because all libraries with GPU support that we analyzed in Chapter 14 target this tool first.

16 API Design

This chapter outlines the design of our Safe-DS API. The design is based on established best practices for API design and general usability, and the analysis of the APIs of existing libraries in Chapter 14. Note that the implementation of the API (Chapter 17) is independent of the design. We may use concepts from library A for the API design but use library B for the implementation.

The chapter is structured in a top-down manner: Section 16.1 contains high-level guidelines that are used throughout the API. Section 16.2 explains the package hierarchy of the API, which groups related API elements together. The subsequent sections then contain details about the individual packages.

16.1 General Guidelines

Our general guidelines are largely aimed at providing *discoverability* (Section 16.1.1), i.e. finding the API element needed to solve a given task, and *safety* (Section 16.1.2) by preventing and detecting errors. All guidelines are essential for *consistency*, too.

16.1.1 Discoverability

Omit rarely needed features. Only functionality that is used sufficiently often in the original libraries should be included. This follows the usability principle of "aesthetic and minimalist design" [26] and improves discoverability of the included functionality [23]. Moreover, it is easier to *add* features than to *remove* them because the latter is a breaking change, while the former is not.

Include features only once by default. Each feature should be mapped to exactly one API element to minimize the API size, unless users expect a feature in multiple places. In the latter case, including the feature in each place can improve discoverability because all users immediately find it, regardless of their mental model [26].

Group API elements by task. Clients use the API to solve a given task. Therefore, the hierarchical structure of the API, which is defined by packages, classes, etc., should be task-oriented. The API reference documentation should also reflect this structure. Users can then easily determine which API elements they must consider.

Prefer methods to global functions. Methods and global functions are largely interchangeable. There are, however, several reasons to map features to methods instead of global functions from a usage perspective[46]:

- *Chaining instead of nesting*: Method calls can be chained (e.g. `obj.a().b()`), while calls of global functions must be nested (e.g. `b(a(obj))`). The former can be written and read naturally from left to right, while the latter must be written carefully and read inside out.
- *Filtering based on receiver*: Code-completion of IDEs is a key tool for finding the needed API elements [26]. Suggestions are filtered based on the text that users enter, and additional information derived from the context. For method calls, this context includes the type of the receiver. So, the suggestions at the cursor position (indicated by `|`) in `obj.a().|` are already heavily narrowed down based on the return type of the method `a`, before the user even starts typing. Meanwhile, code-completion must consider *any* global API element in a global context.
- *No additional imports needed*: Methods can be used without adding more imports, while each global function must be imported. IDEs can automatically add the required import, e.g. after picking a suggestion of its code-completion. However, in an interactive environment like a notebook, users must still remember to execute the altered imports, or they are faced with runtime errors. We will discuss this further in Part IV.

Prefer named functions to operators. Operators, like `+`, can be overloaded in Python [134], i.e. given special semantics if used with instances of custom classes as operands. This can lead to very short code. However, operators do not show up in code-completion, so users can easily miss them. Additionally, readability can suffer, since they cannot convey their meaning as well as a properly named function.

Say, for example, that we have two objects representing columns of a table, `c1 = Column("a", [1, 2])` and `c2 = Column("b", [3, 4])`. The first argument indicates the name of the column and the

[46] Methods can also offer a cleaner implementation, see e.g. the refactoring *"Replace Conditional by Polymorphism"* [11].

second its values. What should be the result of `c1 + c2`? It could be a pairwise addition, resulting in a column with values `[4, 6]`, or a concatenation of the values, resulting in a column with values `[1, 2, 3, 4]`. In either case, it is not clear what the name of the resulting column should be. Some users might also expect that the result is a table consisting of the two columns. In such ambiguous cases, the API should only offer named functions.

Naturally, sometimes operators can be overloaded without ambiguity and might even be expected by users, like for custom numeric classes. In this case, the API should include named functions (for code-completion) and corresponding overloaded operators (to meet the users' expectations).

Avoid uncommon abbreviations. Names of API elements should contain complete words instead of uncommon abbreviations. While this increases verbosity, it improves readability and reduces context switches because users need not consult the documentation to understand the purpose of an API element (see rule "Recognition rather than recall" in [26]).

Moreover, code-completion becomes more useful, which filters suggestions based on the text that is already written. Typing "remove" for example, would show all API elements with that word in their name, subject to further filtering based on the context. API elements where "remove" is abbreviated to, say, "rem" would not appear. Accepting a suggestion from code-completion inserts the complete name, so longer names are hardly more effort to write than short ones, either.

Usage of abbreviations that are unambiguous and well-known in the DS context is fine, like CSV, min, or max. This prevents names getting so long that users can no longer differentiate them [26].

Place more important parameters first. Python allows specifying parameters in any order by using named arguments when calling a function. However, the parameter order of a function definition is also significant for the documentation. There, users should first encounter the parameters they most likely have to change.

16.1.2 Safety

Do not mutate objects. In an interactive environment, like notebooks, the same code may be run multiple times. Between runs, notebooks store a map from variables to their values (more in Part IV). Now, consider the code `table.remove_column("a")`, which is supposed to remove the column with the name `a` from the table. If the method modified the object `table` in place, running the same code a second time would fail because the column to remove would not exist anymore.

Because of this, methods should not mutate their receivers but return new objects with the changes applied (*immutability*). Users must still be cautious to assign the result of the call (or the call would do nothing) and to assign it to a *new* variable because `table = table.remove_column("a")` would have the same issue when rerun as the mutation in place. These problems will be handled in Part III.

Specify types of parameters and results. Adherence to static types (called *type hints* in Python [28]) can be checked by linters like mypy without running the program. This eliminates unnecessary execution of long-running DS programs that contain type errors. Furthermore, static types are the basis for various IDE features, including filtering suggestions based on receiver types in code-completion.

Types should be as narrow as possible to catch the largest number of bugs statically. For example, if a parameter accepts the string values `"light"` and `"dark"`, its type should be `Literal["light", "dark"]`, which allows exactly the listed literals, instead of the more general `str`, which allows any string. This is closely related to the *"Literal sets"* precondition from Section 5.1.

Check preconditions of functions and warn/fail early. Type hints cannot cover all preconditions of a function, so we still need checks at runtime. For example, when accessing a column of a table by

name, the table must actually have a column with that name. These runtime checks should happen immediately at the top of a function before running any expensive computations.

Use built-in Python exceptions or custom exceptions. Exceptions raised by underlying libraries should not be passed along to users. Instead, they should be caught, or prevented by equivalent runtime checks before calling into the library. This allows us to show error messages that are consistent, descriptive, and if possible, include a means to resolve the issue. We can also consider all information that is available in our high-level abstractions when building error messages, such as column names.

Avoid `**kwargs`. A function can have a special parameter `**kwargs`[47] that consumes all named arguments for which no parameter with the same name exists [135]. By default, this means that the function signature does not specify which named arguments are legal.

PEP 692 [136] added special types to describe the legal structure of a `**kwargs` parameter, but this is still harder to read than just listing the allowed parameters separately. Moreover, there is no support in docstring formats like NumPyDoc [39] to document a `**kwargs` parameter.

Avoid parameter dependencies. As discussed in Section 5.1, we define a parameter dependency as a case where the semantics of a parameter `A` depend on the value of another parameter `B`. For example, the value of `A` may be ignored if `B` is set to some value. Such preconditions cannot be expressed statically in general, so runtime checks must be added.

Parameter dependencies can be avoided entirely, however: For instance, we might want to load tabular data from CSV and JSON files. We could model this with a single function that infers the file type from the extension of the `path` parameter. But in the case of CSV files, we must allow users to configure the separator between values, like commas or semicolons, so the data gets parsed properly. Unfortunately, such a parameter is completely irrelevant for JSON files. A simple solution is to provide two functions instead, one to load CSV files and another to load JSON files, where each has an appropriate signature.

Mark optional parameters as keyword-only. By default, arguments can be mapped to parameters by their position or their name. Parameters can also be marked as position-only [75] (can only be passed by position) or keyword-only [76] (can only be passed by name).

The keyword-only variant is useful for long lists of optional parameters with similar types. Forcing users to specify the parameter name, prevents them from accidentally switching two arguments with the same type. Moreover, we can change the order of keyword-only parameters or insert new ones without breaking client code. This is useful for the rule "Place more important parameters first".

16.2 Package Structure

As outlined in the general design guidelines (Section 16.1), API elements should be structured based on their task. Packages, i.e. separate namespaces for global API elements, are the main means besides classes to achieve this in Python. Figure 14 shows the package structure of the Safe-DS API. At the top level, the `data` package contains all API elements for data preparation and visualization. The `ml` package provides features for machine learning.

The `data` package is then split further based on the kind of data that clients want to process. `data.tabular` (Section 16.3) is for tabular data, including time series. `data.image` (Section 16.4) is for image data, be it singular images or lists of images. In each case, the `containers` subpackage contains classes to represent the data.

`data.tabular.transformation` includes classes that learn transformations from data and can apply them repeatedly, akin to the `Transformer`s of scikit-learn (Section 14.5). `data.tabular.plotting` (Section 16.5) handles visualization of tabular data.

[47] The name `kwargs` is frequently used, but arbitrary. Only the double asterisks matter.

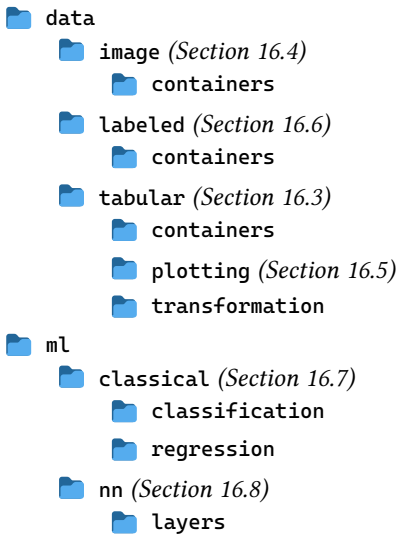


Figure 14: The task-oriented package structure of the Safe-DS API.

`data.labeled` is the bridge to the `ml` package and provides containers that connect features and labels. Instances of these classes are used as input for supervised ML models.

The `ml` package is separated into a subpackage for classical ML (Section 16.7) and neural networks (Section 16.8). This split is necessary because classical models can be segregated further based on the task they solve (e.g. classification and regression). For neural network, the task they solve is already implicitly defined by their layer architecture, so there is little reason to further split this package.

16.3 Data Preparation for Tables (Including Time Series)

A *table* is a rectangular data structure that can be sliced vertically into *columns* and horizontally into *rows*. *Cells* form the intersection of a column and row. Columns are typically named and contain homogeneous data. Figure 15 shows a basic example.

Name	Age
Emil Brandeis	48
Margaret Graham	19
Lewis Hickman	32

Figure 15: A simple example table, with columns (red), rows (blue), and cells (purple).

These terms are well-known and are, thus, used as names of the container classes for tabular data in the Safe-DS API, instead of the terms `DataFrame` and `Series` established by pandas and Polars. In particular, those libraries do not agree whether a `Series` should represent *columns and rows* (pandas) or *only columns* (Polars), so using another term avoids confusion.

`Table`. This class represent an entire table. Instances can be created via static methods, like `from_csv_file` to parse column names and data from a CSV file. Additional methods are provided to write a table back to a file, such as `to_csv_file`. A table can also be constructed from and converted to a list of `Column`s (see below).

Attributes provide access to basic information about the table, like the number of rows (`row_count`) and columns (`column_count`), the names of its column (`column_names`), and its schema (`schema`), which is a mapping from column names to column types.

The method `summarize_statistics` computes statistics, such as minimum and maximum, for all columns of the table at once to quickly understand the data. A similar method is offered by pandas

and Polars, but we also include quality metrics for each column with respect to using it as a feature when training an ML model. This includes, for instance, the *missing value ratio* (number of missing values divided by number of rows) and *idness* (number of distinct values divided by number of rows). A column with a high missing value ratio may not contain enough information to be useful for training. A string column with a high idness, e.g. the names of people, may cause a model to only learn a mapping from this column to the labels, so it does not generalize.

The remaining methods manipulate the table and return the result as a new `Table` object. `Table` objects are immutable, like all other objects in the API (Section 16.1.2). All methods clearly indicate in their name whether they operate on columns or rows. There is no `axis` parameter, like in pandas. The method `remove_columns`, for example, removes columns by their name. There are also some convenience methods, like `select_columns`, which keeps only the listed columns and removes all others. These features can be expressed by combining other API elements, but have proven to be needed often enough to warrant a shortcut.

Some operations must compute a value for each row, like when removing rows that match some query. The corresponding methods of `Table` provide a consistent functional interface: Clients specify a function, that takes a `Row` (see below) as input and returns the computed value. This function is then passed as an argument to the method of `Table`. Functions that take functions as input are referred to as *higher-order functions*[48], while the function passed as input is a *callback*. For example, the following code removes all rows with `19` in the column `"Age"` by using the higher-order function `remove_rows`:

```
table.remove_rows(
  lambda row: row.get_cell("Age") == 19
)
```

`Column`. A `Column` represents a single column and can be created from a name and a list of values with the same type. Attributes contain basic information, like the name of the column (`name`), its number of rows (`row_count`), and its type (`type`). Most methods compute various statistics, like maximum (`max`) or minimum (`min`). Similar to a `Table`, the method `summarize_statistics` can be used to quickly compute a subset of the statistics at once.

`Column` also has higher-order methods that must be called with a function that takes a `Cell` (see below) as input and returns a computed value. The following code, for example, returns a new `Column` with boolean values indicating whether the corresponding value of the original column `age` was equal to `19`:

```
age.transform(
  lambda cell: cell == 19
)
```

`Row`. A `Row` contains named, potentially heterogeneous values. Clients cannot instantiate the class directly, but only receive instances via the parameters of callbacks passed to higher-order functions like `Table.remove_rows`. Expressiveness is not affected by this omission, since clients can simply instantiate a `Table` object with a single row if that is their complete data.

The reason for this is related to the implementation: pandas and Polars both store tabular data in a columnar format, so splitting a table into columns is fast, but splitting it into rows is slow[49]. While our goals (Chapter 15) state that we focus on the API first, the performance when working directly with rows was so poor that usability suffered drastically. In experiments, we observed a slowdown by up to three orders of magnitude between vectorized operations on a `Table` and manual looping over a list of

[48] This term is also used for functions that return functions, but the Safe-DS API does not have these.

[49] If data was stored in a row-based format, the opposite would be true. Such a representation would forgo the possibility to vectorize operations on columns, however.

rows[50]. Hence, we decided to remove the option to initialize `Row` objects, so clients do not accidentally write extremely slow code. We will discuss in Chapter 17 how we still made the `Row` class fast.

The API of `Row` is minimal: The number of columns (`column_count`), column names (`column_names`), and schema (`schema`) can be accessed via attributes. Moreover, the value in a column can be accessed by the method `get_cell`, which returns a `Cell` (see below).

`Cell`. A `Cell` stands for a single value. Clients cannot instantiate this class, and only get instances via parameters of callbacks. This has no effect on expressiveness, since clients can instead instantiate a `Column` object with a single value if this is all their data. Like for `Row`, the reason is efficiency: Working with potentially millions of `Cell` objects is very slow and has a large memory overhead. Chapter 17 is going to discuss, how the `Cell` class is implemented efficiently.

The interface of `Cell` is designed so `Cell` objects behave similar to the values they wrap. When working with cells that contain numbers, which is the most common case, the `Cell` abstraction is fully transparent. In particular, Python operators like `==`, `<`, or `+` are overloaded, so they accept `Cell` operands. The following code, for instance, checks whether the cell value is equal to `19`:

```
1 cell == 19
```

Operations on other types are accessible via namespace attributes: The `dt` namespace contains operations on temporal data. The `str` namespace contains operations on string values. For example, the following code checks whether a string value ends with `"ham"`:

```
1 cell.str.ends_with("ham")
```

Calling an incompatible method on a cell, like a string method on a numeric cell, raises runtime errors. In Python, we do not know the type of the wrapped value statically. This will be improved upon in Section 25.5.

We could implement different subclasses like `NumberCell`, `TemporalCell`, and `StringCell`, each with their own interface, instead of a single `Cell` class that combines all interfaces. However, Python is dynamically typed, so clients can just assume that they have a `StringCell` and call string methods. If this fails, runtime errors are raised. When using static type checkers like mypy, clients would have to add verbose runtime type checks before they can call any type-dependent method on cells. They would also have to define fallback behavior if the type check fails, which would typically be raising a runtime error. In either case, there is no safety gain.

The `Cell` abstraction is largely inspired by Polars's `Expr` class. However, unlike `Expr` it can be mapped directly to a domain concept. Moreover, in Polars, an `Expr` only works on a `DataFrame`. `Series` instead provides a subset of the methods of `Expr` to transform its values. We replaced all those methods in our `Column` class by a higher-order `transform` method. Just like for `Table`s, clients can then specify the transformation of a `Column` via the `Cell` interface. This improves consistency and reduces API size.

`TableTransformer`. Besides the containers for tabular data described above, the Safe-DS API also provides classes that learn a transformation from a `Table` and can apply it to other `Table`s, e.g. for scaling numerical data, encoding categorical data as numbers, or imputing missing values. All classes inherit directly or transitively from the abstract class `TableTransformer`. Subclasses must provide

- an attribute `is_fitted` that indicates whether the transformer has been trained already,
- a `fit` method that learns the transformation from a given `Table`, and
- a `transform` method to apply the learned transformation to a given `Table`.

[50] https://github.com/Safe-DS/Library/issues/575

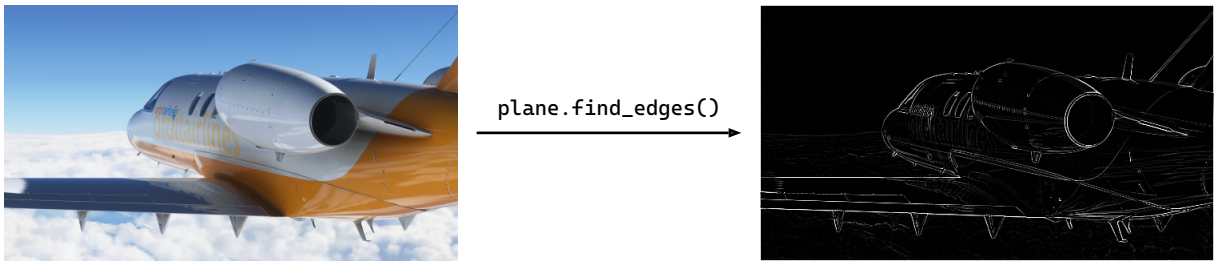


Figure 16: Finding edges in an image via a method of the `Image` class.

`TableTransformer` also has a template method [24], called `fit_and_transform`, that fits the transformer on a `Table` and immediately applies the transformation to the same `Table`, by calling the `fit` and `transform` methods of a subclass.

Furthermore, the API has an abstract class `InvertibleTableTransformer`, which additionally forces implementors to provide a method `inverse_transform` to undo a transformation and restore the original data.

The abstract classes closely mirror scikit-learn's `Transformer` interface. However, since they take `Table` objects as input, they have access to column names and can, thus, provide better error messages to improve safety. The Safe-DS API offers fewer concrete `Transformer` classes than scikit-learn and fewer parameters for each of them to reduce API size. The decision of what to include was based upon usage data, as explained in Section 4.1 and Section 4.2.

16.4 Data Preparation for Images

The Safe-DS API has two abstractions for image data: The `Image` class represents a single image, and the `ImageList` class multiple images.

`Image`. `Image` objects can be created from a file or an in-memory byte array via static methods and written back to a file via instance methods. Basic information, like width, height, and number of channels (e.g. three for RGB images) is accessible via attributes. The remaining methods offer various image manipulations, e.g. to find edges as shown in Figure 16. They each return a new `Image` object. Overall, the interface is similar to Pillow.

`ImageList`. An `ImageList` is a container for multiple images, which may have varying sizes. Clients can access the sizes of the contained images with attributes. The class also has methods to add, remove, and access individual `Image` objects. Additionally, there are methods to remove duplicate images or images with certain sizes. For image manipulation, an `ImageList` offers the same methods as a single `Image`, like finding edges (Figure 16), but applies them to all contained images at once. The result is returned as a new `ImageList`.

Neither of the libraries we investigated in Section 14.3 has an abstraction like `ImageList`. Pillow and scikit-image have no batching at all, and batching via tensors in TorchVision and TensorFlow requires that all images have the same size.

16.5 Data Visualization

For our initial API design, we focussed on static visualizations because we already have an abstraction to represent them: the `Image` class. Interactive visualizations may be added at a later date. The API includes plots for single columns in the `Column` class and for two or more columns in the `Table` class.

Plots for `Column`. A `Column` can be plotted by accessing methods in its `plot` namespace. For example, the distribution of numeric values can be visualized by a histogram, which is created by the call `age.plot.histogram()`, where `age` is some `Column` object. A possible output is shown in Figure 17.

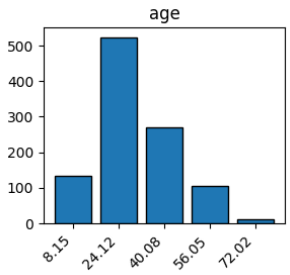


Figure 17: A histogram created for a single `Column`.

All plot methods return an `Image` object, which already has methods for writing to a file, resizing[51], or other manipulations. Styling options are minimal since the visualizations are meant to help understand the data quickly and not for, say, publications. It is only possible to switch between a light and a dark theme, so created plots do not look out of place when viewed in an IDE.

Plots for `Table`. Plots for an entire `Table` fall into three broad categories:

1. Plots for a single column (see above) that are created for multiple columns in the `Table` at once and arranged in a grid.
2. Plots that show the relation between two columns, e.g. a line or scatter plot.
3. Plots that show the relation between multiple column pairs at once, e.g. a heatmap that shows the correlation between all pairs of numerical columns.

Corresponding methods are available in the `plot` namespace of the `Table` class. For example, the code `table.plot.histograms()` plots histograms for all columns of the table and produces output similar to Figure 18. As for `Column`, these methods return an `Image` object and support light and dark themes.

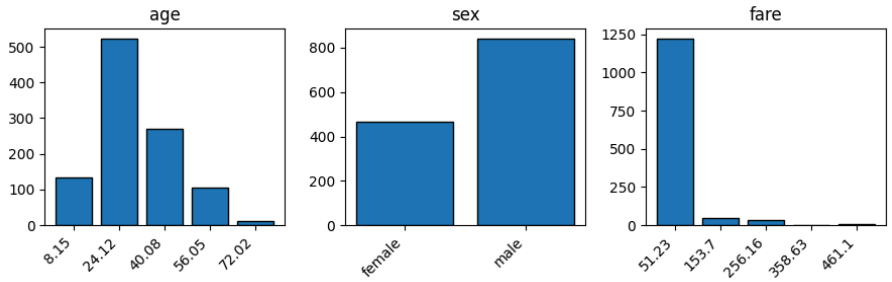


Figure 18: Histograms created for all columns of a `Table` at once and arranged in a grid.

16.6 Containers for Labeled Data

In addition to the raw data, supervised ML models need to know which part of the data to use as features and which as the target to predict. The Safe-DS API includes classes for predictions on tabular data (`TabularDataset`), time series (`TimeSeriesDataset`), and images (`ImageDataset`), which will be explained in this section.

Unlike the dataset classes in NN libraries like PyTorch (Section 14.6), which only define general interfaces and must be subclassed by clients, our dataset classes relate directly to the task to solve and handle all required conversions of the data, e.g. to tensors, internally.

`TabularDataset`. A `TabularDataset` is used if a single target column should be predicted from a list of feature columns. An extension to allow multiple targets is planned. It can be constructed from a `Table` object (Section 16.3) by specifying the name of the target column. Models receive a `TabularDataset`

[51] Since images are rasterized, scaling them up leads to blur. Scaling down plots can be useful, however.

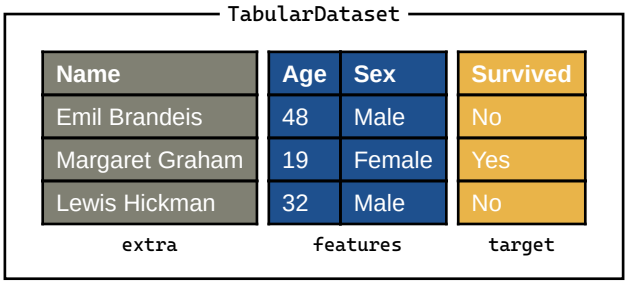


Figure 19: Constructing a `TabularDataset` from a `Table`. ML models predict the `target` column based on the `features`. The `extra` columns are ignored by models during training. During inference they are included unchanged in the output to associate predictions with identifiers.

as input during training and return a `TabularDataset` during inference. In the latter case, the `target` contains the prediction.

Additionally, columns can be marked as *extra*, to indicate they are neither feature nor target. This is useful for columns containing identifiers, like a person's name, as shown in Figure 19. Normally, such columns would have to be removed before training a model. However, then it is difficult to interpret the model's predictions. If the columns are marked as extra instead, models ignore them during training, but still accept them during inference and include them unchanged in their prediction output.

`TimeSeriesDataset`. A `TimeSeriesDataset` also splits a `Table` into features, target, and extra columns. Moreover, it contains configuration that is specific to training on a time series, like the *window size*, which is the number of consecutive samples to use for prediction.

`ImageDataset`. The `ImageDataset` supports three use-cases so far that each use an `ImageList` (Section 16.4) for the features but have different target types:

- To learn a mapping from images to a single value, like the class of a depicted object, the target can be a `Column`. Figure 20 shows a regression example instead.
- To learn a mapping from images to several values, like the counts of various objects, the target can be a `Table`.
- To learn a mapping from images to images, the target can be another `ImageList`.

In any case, the images in the `ImageList` must all have the same size, which is checked upon construction of the `ImageDataset`. This constraint is imposed by the ML models that train on an `ImageDataset`. `ImageList` provides methods to either resize or crop the images to unify sizes.

16.7 Classical Machine Learning

The design of the API elements for classical ML is inspired by the `Predictor` interface of scikit-learn (Section 14.5): Models are represented by classes, which *transitively* inherit from the abstract class `SupervisedModel`. Subclasses must offer

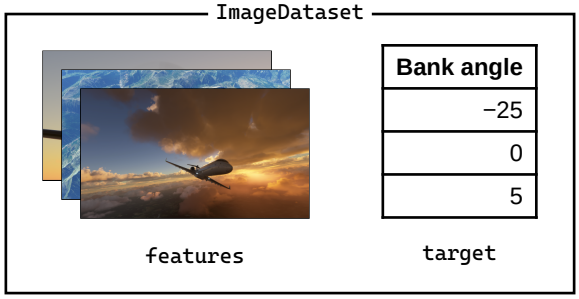


Figure 20: An `ImageDataset` that contains an `ImageList` as features and a `Column` as target.

- an attribute `is_fitted` that indicates whether the model has been trained,
- a method `fit` that takes a `TabularDataset` (Section 16.6) as input and trains the model,
- a method `predict` that takes a `Table` (Section 16.3) or `TabularDataset` as input and returns the prediction as a `TabularDataset`.

Since `fit` and `predict` work with our high-level abstractions for tabular data, they have access to the complete schema of the training data, rather than just its shape, as for scikit-learn. Hence, error messages can point to faulty columns, e.g. if a model cannot be fitted with categorical data, or if columns are missing when predicting.

Concrete models do not directly inherit from `SupervisedModel`, but instead from the abstract classes `Classifier` and `Regressor`, which in turn inherit from `SupervisedModel`. These abstract classes additionally contain template methods [24] to compute various applicable metrics, like the accuracy for a `Classifier` or the mean squared error for a `Regressor`, based on the return value of the `predict` method of implementors. As a shortcut, `Classifier` and `Regressor` offer a method `summarize_metrics` to compute all suitable metrics with one call. The class hierarchy is summarized in Figure 21.

In scikit-learn, classification and regression metrics are instead mixed in a single package, so inexperienced users must consult the documentation to determine whether a metric is applicable to their task. The main other change compared to scikit-learn is simplification by removing seldom-used API elements, in particular parameters, as explained in Section 4.1 and Section 4.2.

16.8 Neural Networks

The design of the package for NNs is largely influenced by Keras (Section 14.6): Forward, convolutional, recurrent, and other layers are offered as predefined classes, which inherit from the abstract `Layer` class. Complete models are represented by the `NeuralNetwork` class, which is instantiated with a list of `Layer` objects, akin to Keras's sequential API. Building arbitrary layer graphs is a planned feature.

A `NeuralNetwork` is trained by calling its `fit` method with some dataset (Section 16.6) and additional options, like a learning rate. Calling a `compile` method, like in Keras is not required, so usage resembles classical ML models (Section 16.7). The training loop is predefined to not overburden novices.

During the training step, the model remembers metainformation about the training data, like the schema of a `TabularDataset` or the image size of an `ImageDataset`. This information is used when the `predict` method is called to validate the data it receives and to know how the prediction should be

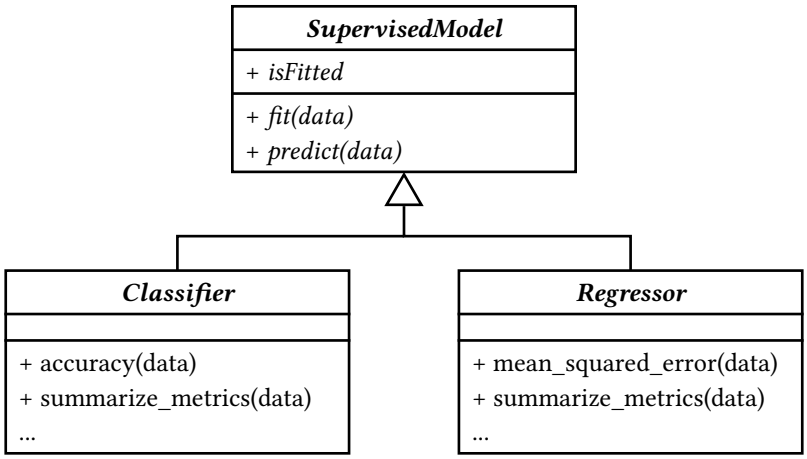


Figure 21: UML class diagram for the hierarchy of the classical supervised models in the Safe-DS API. For brevity, not all metrics methods are shown, and concrete classes, which inherit from `Classifier` or `Regressor`, are omitted.

converted. For example, if a neural network was trained on the `ImageDataset` shown in Figure 20, it would output a human-readable `Column` object, instead of low-level tensors.

Finally, the Safe-DS API provides the option to download a pretrained model from the Hugging Face Hub. For now, these models can only be used unmodified for inference. Fine-tuning models is a planned feature.

17 Implementation (Library)

Acknowledgments. Several students helped with the implementation of the library as part of their Bachelor's theses:

- Simon Breuer [137] implemented the `TabularDataset` class (Section 16.6), and classes for forward layers (Section 16.8).
- Andreas Gerhards [138] implemented the methods to work with temporal data in the `Cell.dt` namespace (Section 16.3), the `TimeSeriesDataset` class (Section 16.6), and classes for recurrent layers (Section 16.8).
- Alexander Fabian Gréus [139] implemented the `Image` and `ImageList` containers (Section 16.4), the `ImageDataset` class (Section 16.6), and convolutional layers (Section 16.8).
- All three also collaborated on the implementation of the `NeuralNetwork` class (Section 16.8).

The Safe-DS API is fully implemented in the Safe-DS library, which is open-source on GitHub[52]. In the remainder of this chapter, we will first explain our choice of backend libraries and how we used the adaptoring approach from Part I to build the Safe-DS library on top of them (Section 17.1). Afterward, we discuss challenges and their resolution when implementing the desired API for containers of tabular data (Section 17.2) and images (Section 17.3) in terms of the API of backend libraries. The remaining elements of the Safe-DS API could be mapped in a straightforward manner to elements of the backend libraries, so we do not discuss their implementation further.

17.1 Reusing Existing Libraries

Choosing backend libraries. Our library is implemented as an adapter library that provides the desired API (Chapter 16), but calls existing libraries internally. The internal libraries are fully hidden from our clients. Using other adapter libraries internally, like seaborn or Keras, might simplify the implementation of our library, but provide no visible benefit to clients. Instead, this would add more dependencies to our library, increasing installation size, so we immediately discarded those options. In the end, we settled on the libraries shown in Table 11 as our foundation.

Getting started using adaptoring. Adaptoring (Part I) proved to be a useful starting point for the Safe-DS library: The usage and usefulness analysis (Chapter 4) highlighted API elements that are required for expressiveness or can be removed for simplicity. The extraction of preconditions (Chapter 5) indicated places where safety could be improved by adding checks or redesigning the API. Further simple but impactful changes like renamings were added in the API Editor GUI (Chapter 7) and initial adapter code was then generated automatically (Chapter 6).

Follow-up manual changes. Afterward, the adapter code was changed directly to apply transformations that are not yet automated or that are too specialized to be worth automating. In particular, new, high-level abstractions like our dataset classes had to be created manually, since they do not exist in the backend libraries. Moreover, glue code between libraries had to be written manually, since it heavily depends on the two libraries to connect.

[52] https://github.com/Safe-DS/Library

Table 11: The libraries we picked as basis for the Safe-DS library.

Category	Chosen Library	Reasons & Remarks
Table preparation (incl. time series)	Polars	• Much better overall performance than pandas.
Image preparation	TorchVision	• GPU acceleration. • Parallel processing of multiple images. • NNs are implemented using PyTorch, so using TorchVision avoids unnecessary conversion.
Data visualization	Matplotlib	• Highly configurable *static* plots. • Other libraries might later be considered for interactive visualization.
Classical ML	scikit-learn	• Wide range of supported algorithms. • The specialized libraries for gradient boosting only offer benefits, if we expose most of their parameters, which is not our goal.
Neural networks	PyTorch	• Better platform support compared to TensorFlow and JAX.

Reducing startup time. Another issue that had to be tackled manually was the startup time of the Safe-DS library: We use several, large libraries in the background, but not all libraries are required for each task. For example, if users want to manipulate tabular data, there is no reason to load PyTorch. Unfortunately, imports in Python can run arbitrary code to initialize a library. Because of this, a simple `import torch` to import PyTorch takes several seconds to complete.

Thus, placing imports of third-party libraries at the top of our Python files, as is usually done, is not an option. A library would be imported eagerly, even if only a single function in our file requires it and that function is never called by clients. As we would integrate more third-party libraries, the startup time of our library would keep increasing. We handle this in two ways:

- Imports of third-party libraries are moved into the functions that need them. Because of this, we must only pay the import cost once the library is needed. Imports in Python are *idempotent*, so calling them multiple times has the same effect as calling them once.
- API elements of our library are always loaded lazily, even if clients import an entire package at the top of their program. The loading is deferred until clients access a specific API element. We use the library `apipkg`[53] for this altered behavior.

Because of these changes, our own code and third-party libraries are only loaded on demand. If something is never needed, it is also never loaded. Hence, more libraries can be added later without affecting the startup time.

17.2 Containers for Tabular Data

We used Polars to implement our containers for tabular data (Section 16.3). Our `Column` is simply backed by a Polars `Series`, which is an equivalent concept. The implementation of our `Table`, `Row`, and `Cell` classes in terms of the classes offered by Polars is more interesting, however, and will be the topic of this section.

`Table`. Polars has *two* classes that are related to our `Table` class, namely `DataFrame` and `LazyFrame`. The latter offers a subset of the `DataFrame` API in a lazy manner, which facilitates further optimizations, as explained in Section 14.1. Polars suggests using a `LazyFrame` whenever possible. Converting a `DataFrame` to a `LazyFrame` is cheap, but the opposite direction is slow in general, because the resulting data must be computed first.

[53] https://github.com/pytest-dev/apipkg

Our `Table` class contains two attributes: `_lazy_frame` contains a Polars `LazyFrame` and is immediately populated upon instantiation. `__data_frame_cache`[54] may contain a Polars `DataFrame` and is only populated if the `Table` is instantiated with a Polars `DataFrame`. Otherwise, it is `None`. Additionally, the `Table` class has a getter `_data_frame`, which populates the `__data_frame_cache` if it is `None` and then returns its value.

The methods of `Table` work with the `_lazy_frame` attribute whenever possible. Otherwise, they use the `_data_frame` getter, which collects the data into a `DataFrame` if needed, and caches it. Clients only notice the difference between lazy and eager methods based on their speed.

`Cell`. `Cell` is an abstract class with one internal implementation called `_LazyCell`. A `_LazyCell` is essentially a builder for a Polars `Expr`. It is initialized with an `Expr` and all its methods modify the `Expr` and return it wrapped into a new `_LazyCell` object. For example, the code in the following two lines is equivalent. It first gets the value in the `Age` column and checks whether it is 19. Notice that `==` is applied on a `_LazyCell` operand in line 1, and on an `Expr` operand in line 2.

```
1 _LazyCell(pl.col("Age")) == 19
2 _LazyCell(pl.col("Age") == 19)
```

Methods of our `Table` and `Column` classes that work with `Cell`s, extract the contained `Expr` and pass it to appropriate API elements of Polars.

`Row`. As we already explained in Section 16.3, Polars stores data in a columnar format. Hence, slicing the data into rows would require instantiating at least one new object for each row of the table, and then iterating over this list using basic Python loops. This is slow.

But since we decided that users cannot create `Row` objects themselves but only receive them in callbacks, we can use a trick: Our sole internal implementation `_LazyVectorizedRow` contains an entire `Table`, so only one new object with a pointer to the existing `Table` is needed. Getting the column count, column names, and schema of a `Row` just delegates to the corresponding attributes of the contained `Table`.

The only method that differentiates `Row` from `Table` is `get_cell`, which gets a cell by column name. This method first checks whether the column exists and then creates a `_LazyCell` that contains a `pl.col` Polars `Expr` to select the column. Hence, the code in the following two lines is equivalent, where `row` is an instance of `_LazyVectorizedRow`:

```
1 row.get_cell("Age")
2 _LazyCell(pl.col("Age"))
```

All callbacks that receive a `Row` as input return a `Cell`. Methods of our `Table` class that work with `Row`s extract the contained `Expr` and pass it to suitable API elements of Polars.

Taken together, the `Cell` and `Row` classes provide the speed of Polars's lazy and vectorized `Expr` class but hide it behind familiar terms from the domain. Moreover, the check whether a column exists is done eagerly in `Row.get_cell`, while Polars also does this lazily. Hence, we can provide earlier feedback and point to the exact location where an invalid column is accessed.

17.3 Containers for Images

Since we use PyTorch to implement our NN package, it was logical to implement our container classes `Image` and `ImageList` (Section 16.4) using PyTorch `Tensor`s. This avoids unnecessary conversions when using images as input to an NN, or later if an NN generates images. Moreover, the methods to manipulate images were implemented using TorchVision, which can work immediately with PyTorch `Tensor`s, too.

[54] Python scrambles the name of class members if it begins with a double underscore. This prevents outside access.

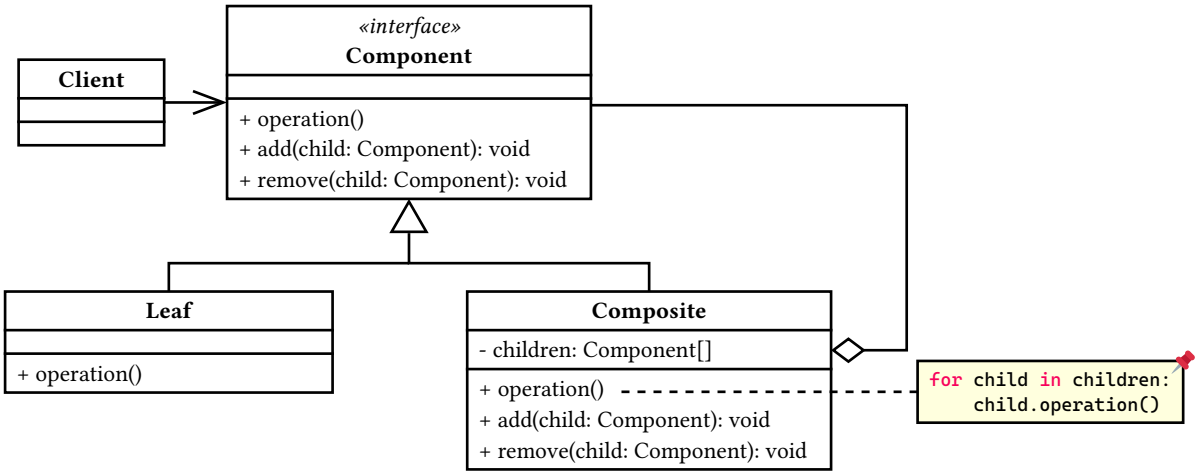


Figure 22: UML class diagram of the composite pattern [24]. The `Client` can treat `Leaf` and `Composite` objects the same and only interacts with them via the `Component` interface.

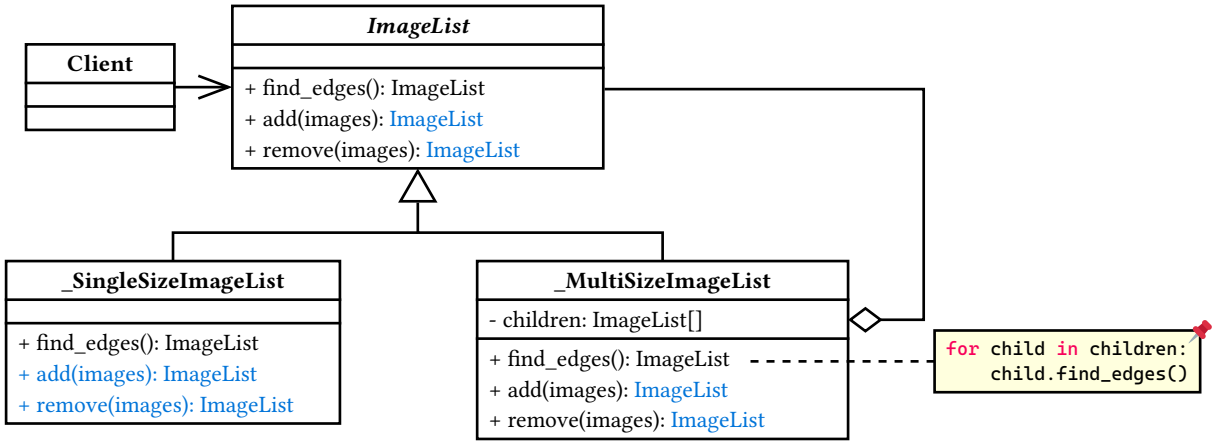


Figure 23: The implementation of the abstract `ImageList` class using a derivation of the composite design pattern shown in Figure 22. Major changes are marked in blue. The `add` and `remove` methods return a new `ImageList` object, which allows them to switch between concrete implementations.

The implementation of the `Image` class posed no further challenge. But, an `ImageList` is supposed to be able to contain images of *multiple* sizes. Implementing an `ImageList` as an actual list of `Images` and looping over them using Python loops would be inefficient, due to the overhead of the additional `Image` objects and the sequential processing with the loop. TorchVision can process multiple images in a batch, but only if they are wrapped into a single `Tensor`. This means that all images must be of the same size.

We solved this issue with a derivation of the composite design pattern [24]. The general structure of the pattern can be seen in Figure 22. `Components` are arranged in a tree, where the `Composite` class represents inner nodes, and the `Leaf` class the leaves. Both implement the `Component` interface, which has methods to change the tree structure and invoke other operations. `Composite` implements these other operations by looping over its children and forwarding the call. Clients generally only interact with the `Component` interface and do not need to know whether they are working on a `Leaf` or `Composite` object[55].

[55]Depending on the implementation of the pattern, the `add` and `remove` methods may only be available on `Composite` objects. In this case, clients must add a type check before they can call these methods.

Figure 23 shows the implementation of our design for image lists. The abstract class `ImageList` takes the role of the `Component` class. It defines methods to manipulate the contained images, like `find_edges()`, as well as methods to add or remove images. The class has two internal subclasses:

- `_SingleSizeImageList` is used for images of the same size. It is backed by a single PyTorch `Tensor` and uses batched operations from TorchVision to manipulate the images.
- `_MultiSizeImageList` can hold images of differing sizes. It contains `_SingleSizeImageList` objects, each containing all images of a specific size.

The `add` and `remove` methods do not manipulate the `ImageList` object that receives the call directly, but instead return a new object. Unchanged internal objects are shared to avoid expensive copying. Returning a new object allows switching between the concrete implementations as needed. For example, adding an image with a new size to a `_SingleSizeImageList` returns a `_MultiSizeImageList`.

The same applies to methods for image manipulation, so calling `resize` on a `_MultiSizeImageList` always returns a `_SingleSizeImageList`. This keeps the data packaged optimally to take full advantage of batching. The switch is transparent to clients, who only interact with the abstract `ImageList` class.

18 Evaluation (Library)

We evaluated the Safe-DS library with two different methods: First, we let several students use the library over multiple sessions to find potential issues, especially regarding expressiveness (Section 18.1). All feedback was immediately acted upon, so the library changed from session to session. Second, we conducted a structured usability study to compare the usability and development speed of the Safe-DS library to the libraries it replaces (Section 18.2). Here, we used fixed versions of the libraries for consistency.

18.1 Informal, Long-Term Usage Experiment

To assess the expressiveness of the Safe-DS library and detect potential learning and usability hurdles, we ran an informal experiment as part of the Bachelor lab "Angewandte Softwaretechnologien 2024". Eight students with varying degrees of DS knowledge used the Safe-DS library[56] to solve tasks on preselected Kaggle datasets[57] that were chosen based on the following criteria:

- The data must consist of tables (including time series), or images. Other kinds of data (e.g. natural language text) are out of scope for the initial prototype.
- The usability score of the dataset must be at least 8, so participants can easily understand the data.
- The size must be at most 500 MB, so participants can comfortably process it on their laptops, even without a GPU.

The experiment ran over six sessions of roughly seven hours each, with one session per week. Feedback was immediately acted upon between the sessions, e.g. by extending the API, to gather further feedback on the changes. The design discussed in Chapter 16 already incorporates these modifications.

Since the library and the level of experience of the participants were constantly evolving throughout the experiment, we do not report usability scores or other metrics like the time required per task. The purpose of this experiment was solely to gather qualitative feedback. Section 18.2 describes how we gathered quantitative feedback as well.

[56] Another eight students tested the Safe-DS language and its IDE, which we will discuss in Section 41.1.

[57] https://www.kaggle.com/datasets

18.2 Structured Usability Study

Acknowledgments. The usability study was conducted by Camilla Grefrath as part of her Bachelor's thesis [140].

The structured usability study was supposed to answer the following research questions:

- **RQ1**: How does the *general usability* of the Safe-DS library compare to the original DS libraries?
- **RQ2**: How does the *learnability* of the Safe-DS library compare to the original DS libraries?
- **RQ3**: Can users of the Safe-DS library *finish more tasks alone* compared to the original DS libraries?
- **RQ4**: How does the *development speed* with the Safe-DS library compare to the original DS libraries?

Participants. 14 Bachelor students in the computer science track of the University of Bonn participated in the study[58]. They all had prior programming experience, but not necessarily with Python. The participants had little to no experience with DS. No participant had seen the Safe-DS library before.

Groups. To prevent a learning effect, the 14 participants were split into two separate groups. Group 1 worked solely with the original DS libraries, and Group 2 with the Safe-DS library. In particular, Group 1 used Polars v1.8.2, Matplotlib v3.9.2, and scikit-learn v1.5.2, while Group 2 used the Safe-DS library v0.29.0. Both groups worked with notebooks in Visual Studio Code (VS Code) v1.96 [141] to ensure that everything was the same but the libraries. The recommended VS Code extensions for Python development were installed[59], e.g. to provide code-completion.

Before the first study session began, each participant filled in a questionnaire, where we asked whether they studied computer science in school, whether they had attended a DS lecture at the university, their perceived level of Python experience, and their perceived level of DS experience. Based on their answers, we separated the participants into groups of seven with an overall equal level of knowledge.

Study design. The study consisted of separate two-hour sessions, where Grefrath acted as moderator and guided a single study participant through a task. Both groups worked on the same task, namely training a model to predict who survived the Titanic accident[60]. We picked this task because the data is easy to understand without specialized domain knowledge. It is also small, so all operations in the pipeline are fast. This avoids frustrating waits, especially as participants were not used to DS.

The task was broken down into several clearly defined steps (here translated from German to English):

- **Step 1:** Data understanding
  1. Load the dataset `task_1.csv` in the file `task_1.ipynb`.
  2. Determine the following information:
     1. Which columns are *non-numerical*?
     2. Which columns contain more than 70% *distinct values*?
     3. Which columns contain more than 70% *missing values*?
     4. Which columns contain more than 20% missing values?
  3. Create a *box plot* for the distribution of the passenger age.
  4. Create a *scatter plot* for the relation between ticket price and number of siblings or spouses.
  5. Create a *heatmap* for the correlation between all numerical columns in the dataset.
- **Step 2:** Data preparation
  1. Remove all columns with more than 70% distinct or missing values.

[58] Seven more students tested the Safe-DS language and its IDE, which we will cover in Section 41.3.

[59] https://marketplace.visualstudio.com/items?itemName=ms-python.python

[60] We originally included a second task, where an NN was trained to differentiate images of cats and dogs. But since the Titanic task already filled the allotted two-hour time slots, we abandoned this plan. Increasing the duration of a session would have also increased the risk of fatigue.

2. Apply a *one-hot encoding* to all non-numerical columns.
3. *Impute* all columns with more than 20% missing values with the *median* of the column.
4. Remove all remaining rows that have missing values.

- **Step 3:** Classification
  1. Split the data into *training data* and *test data* in a ratio of 80:20[61].
  2. Train a *decision tree for classification* on the training data.
  3. Determine the *accuracy* of the decision tree on the test data.
  4. Change your code to limit the maximum depth of the decision tree to 3 and retrain it.
  5. Determine the accuracy of the decision tree on the test data again. Did something change?

The wording for each step was chosen so it had a clear definition-of-done, tested the APIs and not the DS knowledge of participants, and did not point exactly to the correct API elements, so we could test discoverability. The latter principle enabled using the same wording for both groups, too. All DS terms (italicized in the list above) were explained to the participants once they arrived at a step. We also provided a description for the columns in the Titanic dataset as a reference.

Participants were instructed to *think aloud* [63]–[68] while working on the task. During the sessions, voice, and screen were recorded with permission. If participants were stuck on a step, they were first given fixed hints, and then the full solution, if they were still unable to solve it on their own. Afterward, they continued with the next step.

Use of search engines and the documentation of the libraries was explicitly permitted. However, the use of AI assistants was forbidden, so we could test the direct interaction of a human with the APIs.

Result (RQ1): SUS scores. In the end, participants filled in one System Usability Scale (SUS) questionnaire [69], [70] about the usability of the used libraries, i.e. the combination of Polars, Matplotlib and scikit-learn in Group 1, and the Safe-DS library in Group 2.

The original libraries received scores between 32.5 and 65, with a mean of 50.7, standard deviation of 12.1, and median of 52.5. On the adjective scale [72], [73], the mean score corresponds to *poor* usability. The Safe-DS library got scores between 32.5 and 77.5, with a mean of 60.7, standard deviation of 16.3 and median of 62.5. The mean score corresponds to *OK* usability on the adjective scale. Figure 24 shows the distributions in a box plot.

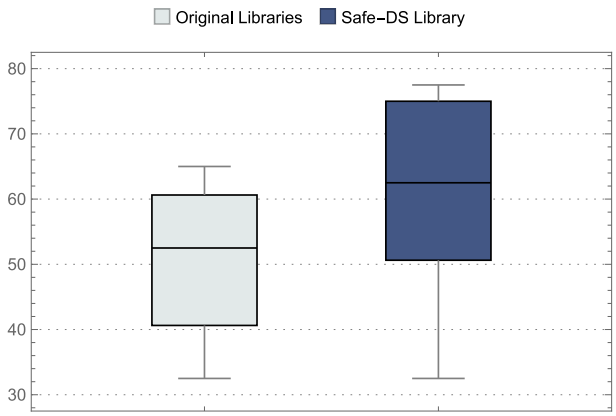


Figure 24: A box plot of the SUS scores for the used libraries. Higher values are better.

[61] We decided to defer splitting the data until here to minimize the number of objects that participants had to deal with in Steps 1 and 2. Splitting sooner would have mostly tested their Python knowledge and not the usability of the libraries.

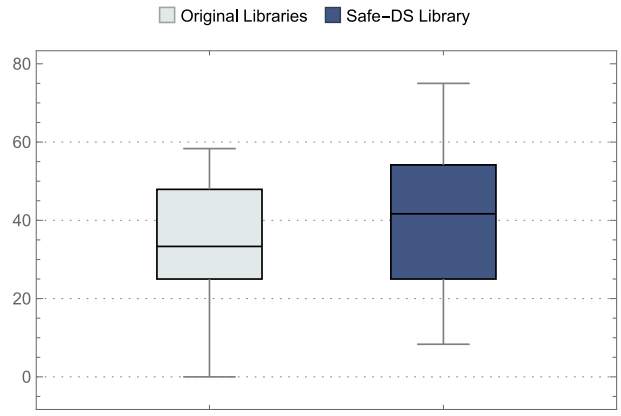


Figure 25: A box plot of the SUS-like scores about the learnability of the used libraries. It is based on two items of the normal SUS questionnaire, and a custom one. The final score is between 0 and 100. Higher values are better.

Result (RQ2): SUS-like scores for learnability. Lewis and Sauro [142] found that the following items in the SUS questionnaire strongly correlate to the learnability of a system:

- Item 4: "I think that I would need the support of a technical person to be able to use this system."
- Item 10: "I needed to learn a lot of things before I could get going with this system."

Interestingly, Item 7 ("I would imagine that most people would learn to use this system very quickly") does not correlate with learnability. The authors conjecture that this happens because people are asked to judge the system for others rather than themselves.

To gather more data points regarding learnability, we also included the following item at the end of the questionnaire, which is meant as a negated version of Item 7 that directly addresses the rater:

- Item 11: "It took me a long time to learn how to use the system."

As for the normal SUS items, participants rated Item 11 between 1 (strongly disagree) and 5 (strongly agree). We then derived a score between 0 (worst) and 100 (best) with the formula $(15 - r_4 - r_{10} - r_{11}) \cdot \frac{100}{12}$, where r_n is the rating for Item n.

The original libraries received scores between 0 and 58.3, with a mean of 33.3, a standard deviation of 19.3 and a median of 33.3. The Safe-DS library had scores between 8.3 and 75, with a mean of 39.3, a standard deviation of 22.4 and a median of 41.7. There is not enough data to compare this result to other systems. The value distributions are summarized in Figure 25.

Result (RQ3): Required help. Participants in Group 1 (original libraries) could not solve 40 out of the 119 steps, but only 2 steps were unfinished in Group 2 (Safe-DS library). Only one person in Group 1 was able to solve all the steps on their own, while only one person in Group 2 was unable to do so. Participants in Group 1 struggled in particular once they had to deal with multiple libraries, i.e. Polars and Matplotlib (Steps 1.3–1.5) as well as Polars and scikit-learn (Steps 2.2, 2.3, 3.2–3.5).

Result (RQ4): Time for the task. Finally, we measured how long the participants took to solve the task, including the third group that worked with the Safe-DS language and IDE (Section 41.3). If a participant was unable to solve a step on their own, we substituted the *mean* time needed by the other participants that did solve the step in all groups[62].

[62]Using the *maximum* time for imputation would be justifiable, too, but only further emphasize the benefits of the Safe-DS library: The mean imputed times would be 144 minutes (Group 1) to 79.71 minutes (Group 2) in this case.

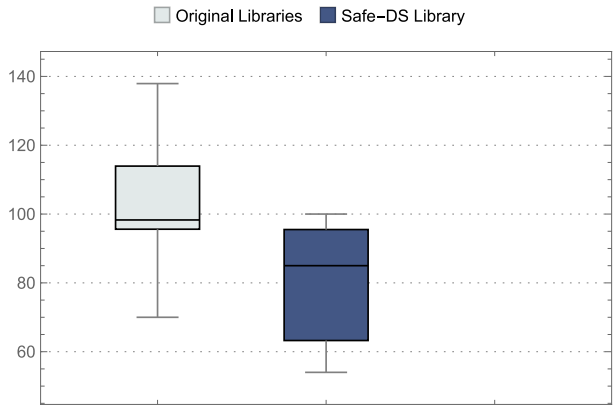


Figure 26: A box plot of the time (in minutes) to solve the task with original libraries (Group 1) and the Safe-DS library (Group 2). If someone could not solve a step on their own, which happened *40 times* in Group 1 and twice in Group 2, we used the mean time that other participants who did solve it required. Lower values are better.

For the original libraries, imputed times range from 70 to 137.9 minutes, with a mean of 102.8 minutes, a standard deviation of 21.0 minutes, and a median of 98.3 minutes. The imputed times for the Safe-DS library were between 54 and 100 minutes, with a mean of 79.5 minutes, a standard deviation of 18.3 minutes, and a median of 85 minutes. Figure 26 shows both distributions as a box plot.

Summary. Compared to the original state-of-the-art libraries, the Safe-DS library

- is more usable (plus 10 points in the mean SUS score),
- is easier to learn (plus 6 points in the mean SUS-like learnability score),
- lets DS novices get considerably more work done on their own (2 vs. 40 incomplete steps out of 119), and
- enables faster development (minus 22.6% mean required time with a generous imputation rule).

18.3 Threats to Validity

Lack of DS, Python, and VS Code knowledge. Besides the libraries, participants had to quickly learn DS concepts, Python, and VS Code, which likely negatively impacted the SUS scores and certainly the times per step. We explained DS concepts when participants arrived at a step, and briefly discussed notebooks in VS Code at the start of each session. Any questions about DS concepts were immediately answered. Problems with Python and VS Code were part of the hints, so we could compare the results with the group that used the Safe-DS language and its IDE (Section 41.3).

Since participants in Group 1 and Group 2 received the same help, had to work on the same task and with the same tools besides the libraries, and had similar overall levels of knowledge, the comparison between the groups regarding our research questions is still valid.

Knowledge of original libraries. Some participants had limited prior experience with the original libraries, so they might have already known some of their quirks that were not revealed in the study. This could negatively impact the SUS scores. Since nobody had used the Safe-DS library before, it was certainly unaffected by this potential bias.

However, typically lack of experience tends to *reduce* SUS scores by up to 16% [74], rather than increase them. So, if anything, the reported SUS scores for the Safe-DS library are too conservative. The results for RQ3 and RQ4 also show objectively that participants had an easier time with the Safe-DS library.

Moderator bias. The main reason the author of this thesis did not conduct the study himself was to prevent biasing participants towards liking *his tool*. Grefrath only told participants that the study

was about testing various libraries for DS, but not who developed them. She conducted each session according to a detailed script, so all participants received the same information and help.

Expressiveness. The individual steps in the task were chosen so they could be solved with both sets of libraries because we wanted to test the *designs* of the APIs and not their *expressiveness*. Naturally, the Safe-DS library is less expressive, since simplicity was one of our explicit goals (Chapter 15).

The study indicates that steps that can be solved with both sets of libraries are easier and considerably faster to finish in the Safe-DS library. Thus, we conjecture that we can add missing but frequently needed API elements to our library in a manner that corresponds to our design goals, without a major negative impact on the answers to the research questions.

Due to the use of methods instead of global functions (Section 16.1.1), the number of API elements in the global namespace stays low. Even the introduction of major new features, like support for natural language text, has little effect on the existing API.

19 Future Work (Library)

Future work revolves around various extension to the Safe-DS API, so it becomes even more widely applicable. The package structure (Section 16.2) was already designed with such additions in mind:

New data containers. Currently, data must fit into a table or consist of images. We could additionally handle other kinds of data, like natural language text or linked data. Support for each new kind of data would be added in a sibling package next to `data.tabular` and `data.image`.

New dataset classes. Our dataset classes tell a supervised model during training which data to use as features and which as the target to predict. Additional combinations could be added in the future, e.g. to map images to images or text to images. These new classes would be added to the `data.labeled` package.

New ML areas. Only supervised ML algorithms are included so far. More classical ML algorithms, e.g. for clustering, could be added in subpackages of `ml.classical`.

Interactive plots. So far, plots are static images that are created with Matplotlib. We could integrate other libraries like plotly that can create plots with useful default interactivity, like showing exact values on hover.

More shortcuts. Finally, API elements could be added to reduce boilerplate code that clients have to write for typical tasks. For example, clients may want to train a model with various combinations of hyperparameter values and pick the best one according to some metric (*hyperparameter optimization*). Manually tweaking the code, rerunning it, and keeping track of the achieved metrics can quickly become tedious.

Because of this, scikit-learn has the classes `GridSearchCV` to exhaustively try all combinations of user-specified values for hyperparameters, and `RandomizedSearchCV` to try a random subset of the combinations. Both are instantiated with an `Estimator` and a description of the values to try, such as

```
{
    "tree_count": [1, 10, 100],
    "max_depth":  [None, 3, 5],
}
```

The dictionary associates the names of parameters (`"tree_count"` and `"max_depth"`) with the values to try for them. This approach has the same issues as a `**kwargs` parameter (Section 16.1): It is impossible to check statically, whether names are written correctly or whether the values in the list have the correct type. This is especially a problem for composed models, like a `VotingClassifier`, where the name for an inner hyperparameter can quickly become complicated, e.g. `"estimator__forest__max_depth"`.

We are currently working on an API extension that integrates hyperparameter optimization directly into the individual models. In our current design, users can e.g. instantiate a random forest for classification with the values to try for its hyperparameters:

```
RandomForestClassifier(
    tree_count = [1, 10, 100],
    max_depth  = [None, 3, 5],
)
```

This design avoids the checking and composition problems described above, since the values to try are connected directly to the parameters. The model can then be fitted by calling one of the methods `fit_by_exhaustive_search` or `fit_by_randomized_search`. Simon Breuer has provided an initial implementation of this concept in his Bachelor's thesis [137] and we are currently refining the design.

20 Conclusion (Library)

In this thesis part, we presented the Safe-DS API, a *simplified*, *integrated*, and *consistent* API for DS. The API is expressive enough to implement complete DS pipelines that involve tabular, time series, or image data. Data can be prepared (Section 16.3, Section 16.4), visualized (Section 16.5), and used to train supervised classical ML models (Section 16.7) and supervised NNs (Section 16.8).

The design of the API places a particular focus on *discoverability*, i.e. making it easy to find the API elements required to solve a given task, and *safety*, i.e. preventing mistakes entirely or at least detecting them early at runtime and helping users recover by offering useful error messages (Section 16.1). A high level of abstraction of the API provides a close mapping to the users' domain model, which is a cornerstone for discoverability.

The API is implemented in the open-source Safe-DS library for Python (Chapter 17). It forwards calls to established, efficient DS libraries, which greatly decreases development and maintenance effort. The library was initially developed with our adaptoring approach (Part I) and then further improved manually.

The evaluation of the Safe-DS library (Section 18.2) revealed that compared to the original state-of-the-art DS libraries, it

- is more usable (plus 10 points in the mean SUS score),
- is easier to learn (plus 6 points in the mean SUS-like learnability score),
- lets DS novices get considerably more work done on their own (2 vs. 40 incomplete steps out of 119), and
- enables faster development (minus 22.6% mean required time with a generous imputation rule).

Because of this, we believe that the library is a great learning aid for DS novices and a useful alternative to the original DS libraries in scenarios where their full expressiveness is not required.

III A Language for Data Science

This part is a rewritten and extended version of a paper published previously as part of my PhD thesis:

- *Safe-DS: A Domain Specific Language to Make Data Science Safe* [143]

Abstract. Due to the long runtime of data science (DS) pipelines, even small programming mistakes, like typographical errors, can be very costly if they are not detected statically. However, even basic static type checking of DS pipelines is difficult because most are written in Python. Static typing is available in Python only via optional, external linters. These require static type annotations for parameters or results of functions, which many DS libraries do not provide.

In this thesis part, we show how Python DS libraries can be used in a *statically safe* way via the Safe-DS language, a domain-specific language (DSL) for DS. The Safe-DS language consists of two connected parts, a simple *pipeline language* to write DS pipelines, and a *stub language* to safely integrate Python libraries.

The pipeline language contains only the language concepts that are required to implement DS pipelines, making it easier to thoroughly analyze than a general-purpose language like Python. The pipeline language statically catches conventional type errors plus errors related to *boundary constraints* of numbers, and *state constraints* of objects, like when predicting with an untrained model, going beyond the abilities of current Python linters. *Schema constraints* of tabular data are checked by running minimal code. In the pipeline language, types and the information required for constraint checking are *inferred*. Hence, users gain static safety without additional effort on their part.

The stub language specifies the signatures and constraints of API elements of Python libraries, akin to header files in C. Stubs are generated by an automated tool, which extracts information from the *code and documentation* of Python libraries.

Static checking is important in DS. Programs that crash at runtime due to an error that could have been detected at compile-time, like a simple misspelling, are always a waste of time, resources and, ultimately, money. This applies in particular to long-running data science (DS) programs, henceforth called *DS pipelines*. A crash late into a DS pipeline can potentially lead to the loss of several hours of work or expensive cloud runtime. Hence, detecting as many categories of errors *statically*, i.e. before executing a DS pipeline, would be a major benefit for all data scientists, regardless of their level of experience. We conjecture that improved static checking has a high potential to reduce DS development costs.

Static checking is hardly used in DS. Unfortunately, most DS pipelines are written in Python and use Python DS libraries. The Python language had 78% usage among all respondents to the 2022 Kaggle DS survey [102]. Python is dynamically typed by default and static type checking is only offered by external linters, such as mypy [133]. But these linters require static type annotations, also known as *type hints* [28], in the used libraries, which many libraries omit (Section 14.7).

The Safe-DS library (Part II) includes type hints, but DS novices are unlikely to install and configure an additional tool, like mypy, to take advantage of them. Van Oort et al. [144] found that nearly half

the ML projects they analyzed did not properly manage their production dependencies. It is fair to assume that novice DS developers also struggle with their development dependencies.

Moreover, the Safe-DS library still raises runtime errors, e.g. if a numeric hyperparameter is outside the legal range, or if an accessed column of a table does not exist. Catching such errors statically or before running expensive operations in the program, would further decrease development time.

Static checking of Python-based DS programs is difficult. Extending the static checking ability of existing linters for Python might appear like an obvious way forward. However, this is not possible because Python is not well-suited for static analysis in general (see also Listing 6):

- Imports are dynamic in Python and can be deeply nested in conditionals whose conditions cannot be evaluated before runtime[63]. This can make it impossible to statically determine, *which API element* of a library, or in fact *which library* is used. Not knowing which code to analyze makes any further attempt at static analysis pointless.
- A similar effect can happen if client code omits type hints. Lacking the related static type information about the receiver of a function call, static analysis tools cannot determine, which library method gets called. But if we, for instance, do not know which `fit` method gets called in `model.fit(learning_rate=1)`, we can neither check whether it accepts a `learning_rate` argument, nor whether `int` is a legal argument type, let alone whether the value `1` is in the legal interval.

Python is needlessly expensive to analyze. Being a general-purpose language, Python contains language concepts that are typically not used by DS pipelines, such as coroutines [145] for asynchronous tasks. Nonetheless, these concepts must be considered by a thorough static analyzer, which would greatly increase the effort to implement it.

Goal: A statically safe language for DS. Our goal in this thesis part is, therefore, to develop a language that can be used instead of Python when developing DS pipelines. It should provide access to the Safe-DS library (and through it to the established high-quality Python DS libraries), but improve upon the safety issues mentioned above by including powerful, built-in static checking. Additionally, third parties should be able to easily integrate more libraries and specify constraints that are checked statically.

Brief part outline. In Chapter 21, we survey the state of the art regarding tools and approaches that aim to provide static safety during development of DS applications. In Chapter 22, we refine our language design goals, outline our approach, and explain the structure of the remaining chapters.

This thesis part contains no dedicated evaluation chapter. Evaluating the language in isolation would require coding in a plaintext editor or on a whiteboard, which are unrealistic scenarios. Evaluation of the language is therefore deferred to Part V, where we present an integrated development environment (IDE) for the Safe-DS language and evaluate the language and its IDE together in Chapter 41.

21 State of the Art: Static Safety for DS

In this chapter, we look into existing tools and approaches to achieve static safety when developing DS applications.

21.1 Existing Approaches and Tools

Wizards (RapidMiner Go). A *wizard* is a user interface that guides users step-by-step through a process [146]. An example in the DS context is the now discontinued RapidMiner Go [147]. With RapidMiner Go, users upload their data, select the feature and target columns, and choose which models should be trained on the data. The tool aids users with the feature selection by computing quality metrics for all columns, like the ratio of missing values.

[63] Imports could even be replaced by calls to methods of Python's built-in `importlib` package [53].

Since users only interact with the system via checkboxes and input fields, many categories of errors are prevented, like misspelling column names. However, the highly restricted nature of wizards also minimizes their *expressiveness*, i.e. their ability to express concepts at all, regardless of ease-of-use. In the case of RapidMiner Go, for example, users must join their data into a single CSV or Excel file before uploading it to the tool. Tweaking hyperparameters of models is not possible.

We do not mention other wizards for DS, since they all suffer from the same issue regarding expressiveness.

Kotlin DataFrame. Kotlin DataFrame [148] is a library for working with tabular data in Kotlin, akin to pandas and Polars in Python (Section 14.1). Kotlin is statically typed, so libraries must provide static type annotations, and client code gets type checked statically. Thus, Kotlin DataFrame already offers better static safety by default than the equivalent Python libraries.

Additionally, users can manually specify the schema of their `DataFrame`, which is a mapping from column names to column types. An optional plugin for the Kotlin compiler then checks whether operations adhere to the schema.

This plugin can also infer the schema of data loaded from a file during compilation, so users need not specify it manually. However, clients must update their code to specifically target this plugin to take advantage of the inference. Schemas of computed `DataFrame`s, e.g. after adding or removing columns, cannot be inferred.

When used in a Jupyter Notebook [149], an interactive programming environment that remembers the current value of variables in a hidden state after program execution, the Kotlin DataFrame library also fetches the schemas of all computed `DataFrame`s from the hidden state. This is used to create typed column accessors on the fly. If users only interact with columns via these accessors, they can avoid errors related to column names and types. However, the accessors are only available after running the code and can get outdated as the code is changed but not rerun.

MLE. Machine Learning Evolves (MLE) [150] is an external[64] *domain-specific language* (DSL) that offers data preparation for tabular data, and supervised ML models. No prototype is publicly available, so the following description is compiled from [150] without further verification.

The system tracks metadata about objects, like the schema of a table, throughout a pipeline. Where possible, the metadata is inferred without running the full program. For example, the schema of the output table after removing a column is inferred to be the schema of the input table without that column. If the metadata cannot be inferred, the DSL code is compiled to Python and run on a server, from where the computed metadata is sent back to the DSL.

The DSL uses the metadata to show errors and warnings in its integrated development environment (IDE). For example, accessing a column that does not exist will show an error. Moreover, the IDE provides code-completion for column names.

The language is implemented in the Meta Programming System (MPS) [152], [153], a language workbench for the development of external DSLs. DSL programs are written using a *projectional editor*. This means that users alter the abstract syntax tree (AST) of a program directly when interacting with the editor, e.g. by accepting suggestions from code-completion or typing an alias that is associated with a language concept. The abstract syntax tree is then projected to users, in various concrete syntaxes, e.g. as text, mathematical notation, or tables.

[64] An *external* DSL is an independent language. In contrast, an *internal* or *embedded* DSL is a library in a host language and generally constrained to the syntax of the host language [151].

MLE inherits the advantages of its MPS foundation:

- **No syntax errors**: Since users directly alter the AST, syntax errors are avoided.
- **Extensibility**: As users directly modify the AST, language concepts need not have unique projections. For example, two different language concepts can be projected to the same text. This might confuse users, but the AST is still unambiguous. In contrast, the concrete syntax of a textual language must be unambiguous, so a parser can correctly create an AST.

  Because of this, MPS is well suited for language composition, i.e. adding more concepts to an existing language [151]. Language concepts reside in packages to prevent name clashes, so even extension that were developed independently can be combined. Extending a textual language in the same manner would quickly lead to ambiguity when parsing code or require unreadable concrete syntax.

  MLE uses this feature for extensibility. All its operations are implemented as individual language concepts. Third-party developers can then create their own language extensions, which model new operations as additional language concepts, e.g. for neural networks.

However, MLE also inherits the disadvantages of MPS:

- **Unfamiliar editing experience**: Users cannot simply type somewhere, but must move the program from one valid AST to another, e.g. by interacting with the code-completion.
- **Lack of standard tools**: MPS-based programs are always serialized to an XML file, even if they are projected to text. The text is merely a means to display the program on the screen, but cannot be stored on disk. Thus, standard tools for diffing or code review are challenging to use.
- **Tool lock-in**: Programs can also only be reasonably edited in MPS, leading to tool lock-in[65].
- **Huge technology stack**: Although extensibility is possible, using it is very difficult. Adding a single new operation requires intricate knowledge of MPS. A developer must *at least* define
  - the structure of the AST node,
  - an editor that describes how the AST node should be projected to the user, and
  - a text generator that creates executable code for the language concept in a target language.

  Additionally, constraints for validation, integration into the type system of the language, actions for user-friendly editing of the AST, tests for all these aspects, and more might need to be added. Overall, extending MLE is possible, albeit rather complicated.

MetaR. MetaR [154] is an external DSL for the preparation and visualization of tabular data. Like MLE, it is implemented in MPS, so the same remarks regarding the use of projectional editing apply. Operations are provided as individual language concepts. MetaR also includes language concepts that allow users to define the file paths to available tables. Schemas of tables that are backed by files are inferred and may be improved manually. This information is used for checking and code-completion.

Extension is possible via language composition, as for MLE. In addition, functions written in the R language[66] can be integrated into MetaR via so-called *stubs*. These stubs contain the names of available functions, and their parameters. For each parameter, they list their name and default value. Types of parameters and results are not included. All stub functions can be called in a MetaR program via a single, generic language concept. This concept makes the system easier to extend, compared to language composition. It *sacrifices static safety*, however, due to the lack of static type information about parameters and results.

[65] A pipeline written in a textual language can be edited in any text editor, albeit potentially without features like syntax highlighting or code-completion.

[66] https://www.r-project.org/

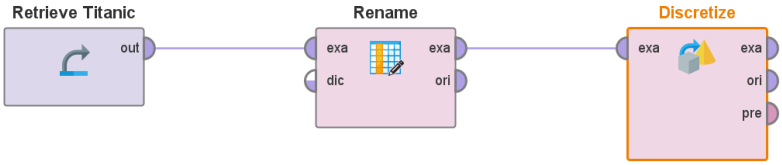


Figure 27: Defining a DS pipeline in Altair AI Studio. Boxes represent individual operations. Half-circles (ports) stand for inputs on the left of boxes, and for outputs on the right. The color of ports visualizes their type. Output and input ports of compatible type can be connected via edges.

Altair AI Studio. Altair AI Studio (formerly RapidMiner Studio, formerly YALE) [155] is a graphical, no-code tool for developing DS applications. Users upload their tabular data into the so-called *repository* of the system. This step infers the schema of the data and computes other metadata, like the number of missing values in each column.

Afterward, users can access data from the repository in a pipeline by selecting the `Retrieve` operation in the system's toolbox and dragging it onto the screen. Altair AI Studio includes additional operations to manipulate the data, train ML models, and more, which are added similarly, leading to a graphical pipeline similar to Figure 27.

Each operation is represented by a box. The input and outputs of operations are shown by ports (half-circles) on the left and right side of operations, respectively. Ports are typed and display their type by their color, e.g. purple for a table or purple/white for a dictionary. Input and output ports can be connected via edges, which defines the data flow in the program. Only ports of compatible type can be connected, which eliminates type errors.

Operations can be configured further by parameters, which are entered in a sidebar (Figure 28). Parameters are typed, and suitable input fields are displayed based on this type, like a number input field for `size of bins`. Legal values of numeric parameters can be constrained to some interval, too. For example, entering a negative number for `size of bins` causes the value to be set to `1` instead, which is the lower bound.

Once the input ports of an operation are connected, the input fields in the sidebar also consider the metadata of the inputs, like the schema of a table. The dropdown for the `attribute` parameter in Figure 28, for instance, only shows the names of numeric columns of the input table.

An operator is implemented as a Java class that extends from the abstract `Operator` class. This approach is used by all built-in operators and can also be used by third party extensions. An `Operator` defines

- a method `doWork` that computes the actual results,
- the fields `inputPorts` and `outputPorts` for ports and their types,
- a method `getParameterTypes` that lists parameters, their types, and additional constraints, and
- a method `modifyMetaData`, which computes the metadata of outputs, such as schemas.

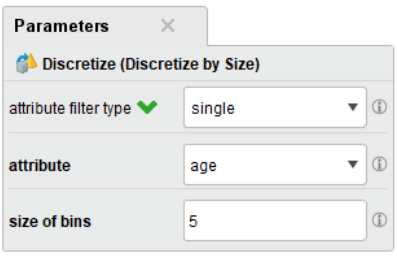


Figure 28: Parameters of the `Discretize` operation that can be set in a sidebar. Input fields are chosen based on the type of the parameter and metadata about the values connected to input ports.

Computing schemas can be as expensive as running the operation, e.g. when removing columns based on the data they contain. In such cases, `modifyMetaData` may also only return a rough relation between the metadata of input and outputs, e.g. that the output includes a subset of the columns of the input.

Overall, Altair AI Studio has the safety that we want, but the fact that the system *only* has a graphical view and is a commercial product has several disadvantages:

- **Slow editing experience**: As experience grows, searching operations in a toolbox, dragging them onto a canvas one by one, connecting them, and then configuring them in the sidebar can be much slower compared to typing out a pipeline.
- **Hidden information**: A graphical view takes up considerably more room than an equivalent textual representation, so it can get cluttered as the pipeline grows. Altair AI Studio partially solves this by hiding parameters in a sidebar that only opens when a user clicks on an operation. However, this is at the expense of never seeing the complete semantics of a pipeline. In Figure 27, for example, we only know that something is renamed, but not what.
- **Lack of standard tools**: Altair AI Studio stores pipelines in proprietary XML files. These are not well-suited for diffing in version control tools, or collaborative practices, like code reviews.
- **Tool lock-in**: Using a graphics-only tool leads to tool lock-in, as nobody is going to edit the serialization directly.
- **Vendor lock-in**: When the tool was still called RapidMiner Studio, its core code was open source on GitHub[67], but the last commit in the public repository is from August 2021. In September 2022, RapidMiner Studio has been acquired and rebranded [156]. Altair AI Studio is still offered for free under a very limited personal license[68], and a full academic license. More restrictive licensing might apply in the future. RapidMiner Go (reviewed first in this chapter) has been completely abandoned after the acquisition.

21.2 Research Gap

Out of the reviewed tools, only Altair AI Studio is as expressive as the already heavily simplified but integrated Safe-DS library (Part II). The rigid wizard interface of RapidMiner Go offers little customization in general, and all tools but Altair AI Studio focus solely on tabular data.

MLE and MetaR compensate for the lack of expressiveness by offering extension points for external developers, but they either need intricate knowledge of MPS and lengthy development, or lack type safety. In Altair AI Studio, new operations can be added via a Java API.

Finally, RapidMiner Go, MLE, and Altair AI Studio offer the greatest gain in static safety out of the analyzed tools. Table 12 summarizes these traits.

While Altair AI Studio ticks most boxes, it is a commercial, *graphical* editor. There is currently no *textual* language that offers the expressiveness and safety of Altair AI Studio. Compared to a graphical tool, a textual language

- can accelerate development if users have programming experience,
- is more compact and shows the entire pipeline at once,
- integrates well with established development tools (e.g. version control systems), and
- is less prone to tool lock-in, particularly versus closed source graphical tools.

Moreover, the textual language can act as the common serialization format for other user interfaces, like wizards, or a graphical editor, to reap their benefits for users without programming experience. We will explore these options in Part V.

[67] https://github.com/rapidminer/rapidminer-studio-modular

[68] For example, tables may only contain up to 10,000 rows.

Table 12: The analyzed tools for improving safety when developing DS pipelines.

Property	RapidMiner Go	Kotlin DataFrame	MLE	MetaR	Altair AI Studio
User interface	Wizard	API + Compiler Plugin	Projectional DSL	Projectional DSL	Graphical Editor
Open source	✗	✓	✗	✓	✗
Expressiveness like Safe-DS library	✗	✗	✗	✗	✓
Extension points for third parties	✗	✗	(✓) language composition	✓ language composition + stubs	✓ Java API
Type safety	✓	✓	✓	(✓) not for stubs	✓
Schema safety	✓	(✓) after code is run	✓	✓	✓

22 Approach: A Textual Language for DS with Two Parts

Goals. To enhance the state of the art, we want to develop a **textual** language for DS, which we call the *Safe-DS language*, with the following properties:

- **Expressiveness**: Developers of DS pipelines should have access to all API elements of the Safe-DS library out of the box.
- **Extensibility**: Third parties should be able to *easily* integrate other Python libraries, so they can be used for pipeline development. The process should be *automated* as much as possible.
- **Safety**: Incorrect use of the Safe-DS library or any third-party library that was integrated by the extension system should be detected as early as possible to avoid costly runtime errors. Ideally, such checks are done statically without running the pipeline at all.
- **Usability**: Implementing DS pipelines should be easy and intuitive. In particular, the safety gain should not increase the effort to write pipelines.

These goals partially overlap: If we offer a mechanism to integrate third-party libraries easily and safely, we can use the same mechanism to include the Safe-DS library. Thus, the expressiveness goal boils down to using the extension concept to ship the Safe-DS library by default.

Two kinds of users. The goals make it apparent that we must deal with the needs of two different user groups:

- *Pipeline developers* use the available Python libraries to implement a DS pipeline for some task. They benefit from the expressiveness, safety, and usability of the language.
- *Library integrators* enable the use of new Python libraries for pipeline developers. They benefit from a simple, yet powerful extension mechanism that allows the specification of static checks for the added API elements. Automation further facilitates their task.

Two language parts. Because of this, we split the Safe-DS language into two parts:

- Pipeline developers use the *pipeline language* (Chapter 23) to write DS pipelines.
- Library integrators use the *stub language* (Chapter 24) to integrate Python libraries.

The pipeline language and stub language are connected by the safety goal: The stub language must provide the required information to statically check the pipeline language. Chapter 25 explains how this is done without impacting the usability of the pipeline language.

23 Writing DS Pipelines in the Pipeline Language

General-purpose or domain-specific language. The pipeline language must provide all concepts to implement DS pipelines with the included Python libraries for DS, like the Safe-DS library (Part II). We must initially determine, whether we need a Turing-complete, general-purpose language with branching (e.g. an `if` statement) and loops (e.g. a `while` statement), or whether a less expressive *domain-specific language* (*DSL*) suffices, which would enable more thorough static analysis.

Let us first look at the state of the art (Chapter 21): For wizards like RapidMiner Go and libraries like Kotlin DataFrame, whether they have branching and loops is not applicable, and MLE does not specify it. MetaR has neither branching nor loops. Altair AI Studio has operators for branching (e.g. `Branch`) and loops (e.g. `Loop`), but treats them as black boxes, like any other operator, i.e. it does not check or visualize what happens inside. The graph view (Figure 27) always shows a directed acyclic graph.

Biswas et al. [3] analyzed the AST nodes of DS pipelines and found that only 4% of them correspond to branching or loops. The loops were mostly needed to define the training loops for NN libraries. They argue that DS pipelines follow a sequential structure.

The Safe-DS library handles the training loop for NNs internally. Its higher-order functions, such as `Table.remove_rows` (Section 16.3), abstract typical tasks and encapsulate branching and loops. Higher-order functions could also be used to implement generic branching or loops, akin to the `Branch` and `Loop` operations in Altair AI Studio.

Because of this, we conclude that the pipeline language does not require branching or loops. These would be needed to *implement* DS libraries, but not to *use* them. Hence, the pipeline language is designed as an *external DSL*, i.e. an independent language that is not embedded into another language.

Identifying the domain. When designing a domain-specific language, the foremost question is the definition of the domain. The pipeline language is supposed to ease development of DS pipelines. Hence, DS is the obvious choice for the domain, as is done by MLE and MetaR (at least for tabular data). In this case, the language would provide dedicated language concepts for DS operations, like filtering the rows of a table, resizing images, creating plots, configuring and training a decision tree, building and training neural networks, evaluating model performance, etc.

This list is by no means complete, but already illustrates that this approach is destined to fail, especially for a textual language:

- Each language concept needs unique concrete syntax, so it can be parsed unambiguously. Trying to support even the operations that are currently included in the Safe-DS library would lead to a bloated language with an enormous set of keywords[69]. Such a language would not be usable.
- A textual language does not support language composition, so all extensions must provide unique concrete syntax, too. This would either make independent development of extensions impossible, or require namespaced keywords (e.g. `com.example.word2vec`) in the concrete syntax, which would be unusable. The design of suitable, user-friendly concrete syntax could not be automated, either.
- Each language concept needs to be considered for static checking (safety), which linearly increases the effort to implement it.

Summing up, modeling the DS domain requires a lengthy development effort to offer the desired expressiveness and safety, and entirely fails to provide extensibility and usability.

Modeling the pipeline domain. There is another solution, though: We initially argued that DS pipelines are sequential and need neither branching nor loops, given that they can access an expressive DS library.

[69] The `Table` class alone has 41 methods in version 0.29.0 of the Safe-DS library.

```
1 package com.example.titanic
2
3 from safeds.data.tabular.containers import Table
4
5 pipeline whoSurvived {}
```

Listing 8: The core structure of a Safe-DS pipeline file. It first states its *package* (line 1), which is the namespace for all declarations in the file. Then, declarations from other packages are *imported* (line 3). Finally, a *pipeline* is defined (line 5), which is the entry point of a program.

We can generalize this concept: A *pipeline* is a sequence of accesses to API elements of a library, with a potential assignment of the results to variables. Variables can be referenced later to access the stored value. Programs that fit these criteria fall into the *pipeline domain*.

Our pipeline language is an *external* DSL that models the pipeline domain, so it is entirely decoupled from the DS domain. The connection to DS is established by the integrated libraries, which form an *embedded* DSL in the pipeline language. Section 23.1 explains the concepts and syntax of the pipeline language in depth, and Section 23.2 briefly discusses its expressive power.

23.1 Concepts and Syntax

Core file structure. Let us now look at the pipeline language in more detail, beginning with the core structure of a file. Listing 8 shows the initial state of a program that aims to predict who survived the Titanic accident. We will use this as a running example.

First, the file denotes its *package* (line 1), which is the namespace for all *declarations* (language concepts with a name) in the file. The use of packages prevents name collisions if developers create Safe-DS files independently, e.g. by using reverse domain names. Multiple files can belong to the same package, similar to Java or Kotlin.

Next, declarations from other files are imported (line 3), so they can be used inside this file. Imports contain a package name (`safeds.data.tabular.containers`) and the name of the declaration to import (`Table`). They are always listed at the top of a file, unlike in Python, where they can be nested in conditionals. This enables the language to statically resolve references to declarations in other files. Declarations in the same package are available without import, as are declarations in any `safeds.*` package. This corresponds to the Safe-DS library, which acts as the built-in library of the language. Hence, the import in Listing 8 could be omitted.

Lastly, Listing 8 contains a declaration of a *pipeline* (line 5), which is the entry point into a program, akin to a `main` method in Java. Next, we will add more code inside the curly braces of the pipeline to define its behavior. For brevity, we will omit unchanged parts of the program.

Statements for sequential execution. The individual actions of a pipeline are described by *statements*. As explained above, the pipeline language only needs sequential control flow, so the language just has two kinds of statements: assignments and expression statements. Both are shown in Listing 9, where a `Table` is loaded from a CSV file and a heatmap showing the correlation between its numeric columns is written to a PNG file.

An *assignment* (line 6) computes an *expression* (its right-hand side), which refers to language concepts that produce results or have side effects, like writing to a file. The results of the expression are assigned to *placeholders*. Some expressions may return multiple results, e.g. when calling `Table.splitRows`, hence the use of the plural. Results and placeholders are matched by index, i.e. the first result is assigned to the first placeholder, etc. The value assigned to a placeholder may be retrieved via a *reference* to the placeholder (line 8) in the code after the assignment.

```
5  pipeline whoSurvived {
6      val rawData = Table.fromCsvFile("titanic.csv");
7
8      rawData
9          .plot.correlationHeatmap()
10         .toPngFile(path = "heatmap.png");
11 }
```

Listing 9: Statements define the actions of a pipeline. An *assignment* (line 6) stores the results of an expression in *placeholders*. An *expression statement* (lines 8–10) executes an expression for its side effects, like writing data to a file but discards its results.

Unlike variables in other programming languages, placeholders cannot be reassigned later. Together with the deep immutability provided by the Safe-DS library (Section 16.1.2), this ensures that each reference to, say, the placeholder `rawData` in Listing 9 refers to the same value throughout a pipeline. We picked these semantics because being able to overwrite placeholders would not increase the expressiveness of a purely sequential language. Instead, we conjecture that using the same name for different values throughout a program would hinder program comprehension. It would also certainly make static analysis more difficult to implement. Finally, our execution concept for the DSL (Part IV) and the graphical view (Section 39.2) depend on this property of placeholders.

Expression statements (lines 8-10) execute an expression, too, but discard any results. They are useful if the expression has side effects.

Expressions. As explained above, expressions produce results or have side effects. Listing 9 already includes various kinds of expressions:

- *String literals* (e.g. `"titanic.csv"`) denote constant strings. Together with the other kinds of literals in the language, e.g. for numbers and Booleans, they are the basic building blocks of expressions.
- *References* point to declarations, which are language concepts that have a name. Syntactically, a reference consists of the name of the declaration. These declarations can be local to the pipeline, such as placeholders (e.g. `rawData`), or be API elements of an external library (e.g. `Table`).
- *Member accesses* point to declarations inside a class of an external library (Section 24.1). They can either be static members accessed via the class itself (`Table.fromCsvFile`) or instance members accessed via an instance of the class (`rawData.plot`).
- *Calls* pass *arguments* to a *callable*, run the code inside the callable, and return its *results*. Callables may be defined as segments in the pipeline language (see below), or be functions of an external library. Arguments may be passed by position (e.g. `fromCsvFile("titanic.csv")`) or name (e.g. `toPngFile(path = "heatmap.png")`). The nth positional argument is assigned to the nth parameter. A named argument is assigned to the parameter with the same name. Named arguments may only be listed after positional arguments, and the same parameter may not be assigned multiple times.

Member accesses and calls can be *chained* (lines 8-10 in Listing 9), which means they are applied directly to the result of a prior expression, instead of assigning that result to a placeholder first. The assignment to the placeholder `rawData` in Listing 9 could also be replaced by chaining.

The pipeline language has more kinds of expressions, like arithmetic, comparison, and logical operations, but these are well-known from other programming languages, so we refrain from covering them here. The documentation for the pipeline language contains a complete list [157].

Injecting behavior. The Safe-DS library has higher-order functions, e.g. to remove rows of a table that match some query (Section 16.3). Therefore, the pipeline language also needs a means to specify callbacks. In Python, this is typically done by *lambdas*, which are unnamed callables that compute a single expression. A Python lambda cannot contain statements [135]. We decided to eliminate this

```
10      // ...
11
12      val blockLambda = rawData.removeRows((row) {
13          val age = row.getCell("age");
14          yield shouldRemove = (19 < age) and (age < 25);
15      });
16
17      val expressionLambda = rawData.removeRows(
18          (row) -> row.getCell("age") == 19
19      );
20  }
```

Listing 10: Calling higher-order functions with *lambdas* as arguments. *Block lambdas* (lines 12–15) can contain multiple statements and specify their results using assignments and the `yield` keyword. *Expression lambdas* (lines 17–19) are syntactic sugar and compute a single expression.

restriction for our lambdas, so common parts of a complex expression can be extracted. Listing 10 demonstrates this in lines 12 to 15, where a cell is first assigned to a placeholder and then referenced twice to compute the result of the lambda.

Such a lambda is called a *block lambda*. It specifies its results using assignments with the keyword `yield` instead of `val`, as seen in line 14. Results of block lambdas are named, which aims to improve documentation. A block lambda may yield multiple results and, unlike a `return` in other languages, `yield` does not end execution of the lambda code[70].

As syntactic sugar, the pipeline language also has *expression lambdas*, which compute a single expression and return its result. These can be more concise than block lambdas in simple cases (Listing 10, lines 17-19).

Lambdas are *closures* [158], i.e. declarations that are available in the scope that the lambda is created in are available in the body of the lambda, too. For example, we could reference the placeholder `blockLambda` in the body of the expression lambda in line 18 of Listing 10. As the pipeline language does not support reassignment, we need not define whether lambdas store a *copy* of the referenced values or maintain a *live* pointer.

Lambdas must only be passed as arguments to named callables. In particular, lambdas may not be assigned to placeholders and called later, passed as arguments to other lambdas, or yielded by other lambdas. This suffices to integrate the Safe-DS library, and (deliberately) reduces the expressive power of the language, as we will see in Section 23.2, making it easier to statically analyze. The restriction leads to a simpler graphical representation for Safe-DS pipelines as well (Section 39.2).

Extracting and reusing DSL code. Certain operations in a pipeline must be repeated on different values, e.g. to clean training and validation data. To prevent code duplication, such code can be extracted into a *segment*[71] (Listing 11, lines 24 to 28).

Segments are named callables that define their parameters (`data`) and results (`cleanedData`) as part of their header (line 24), which includes their type (`Table`). Parameters and results of segments are the only place in the pipeline language where *manifest types* are needed, i.e. where pipeline developers have to specify types in their code. The body of a segment contains the statements that compute its results. Similar to block lambdas, results are set by using assignments with the keyword `yield`.

Segments can be called inside of pipelines, lambdas, or other segments (line 21). A segment may not call itself directly or transitively, as this leads to infinite recursion due to the lack of conditional execution.

[70] This would be pointless as the pipeline language has no conditional execution.

[71] This term is non-standard, but fits the pipeline metaphor.

```
    // ...

    val trainingData = cleanData(rawData);
}

segment cleanData(data: Table) -> cleanedData: Table {
    yield cleanedData = data.removeRows(
        (row) -> row.getCell("age") == 19
    );
}
```

Listing 11: Common code can be extracted into a *segment* and called multiple times in a pipeline.

23.2 Expressive Power

We now want to discuss whether the pipeline language is Turing-complete, which has implications on the ability to statically analyze programs written in it, e.g. for type inference (Section 25.3). Rice's theorem [159] implies that no algorithm can be constructed that can decide whether a program written in a Turing-complete language has some non-trivial semantic property. Here, non-trivial means that some programs have this property whereas others do not. A semantic property relates to the behavior of a program (e.g. does it terminate for all inputs?), rather than its syntax.

The pipeline language is purely sequential and contains neither loops nor conditional jumps, so the statements are insufficient to emulate a Turing machine. But it includes lambdas, which could be used to emulate the lambda calculus [160], which would also imply Turing-completeness. Here, arbitrary computations are represented solely as lambda terms, which are built from the basic rules in Table 13.

For example, we can represent the Boolean `true` by the term $\lambda a.\lambda b.a$ and `false` by $\lambda a.\lambda b.b$. Then we can encode the Boolean disjunction as $\lambda x.\lambda y.((x\ \lambda m.\lambda n.m)\ y)$. Using the application rule twice on the disjunction term to set x to the lambda term for `true` and y to the lambda term for `false`, and simplifying the term (β-reduction), correctly yields the result $\lambda m.\lambda n.m$. This is equivalent to the lambda term for `true`, since variable names are not relevant in the lambda calculus (α-equivalence). A truth table proves that the definition works for other arguments, too, but we omit this here.

The pipeline language cannot emulate the lambda calculus, though, as lambdas may only be used as arguments when calling named callables. In particular, callables (including lambdas) must not yield lambdas (so abstraction is not possible), and lambdas must not be passed as arguments to other lambdas (so application is not possible).

Because of this, the pipeline language itself is not Turing-complete[72]. This simplicity is by design to facilitate thorough static analysis of code written inside the language, as we will explain in Chapter 25. The language still gains the required expressiveness from the high-level, domain-specific operations provided by libraries, which is the topic of Chapter 24.

Table 13: The three rules to build terms in the lambda calculus.

Name	Syntax	Description
Variable	x	Some function parameter, which can be an arbitrary character or string.
Abstraction	$\lambda x.M$	A function definition with one argument x that returns the body M, which is an arbitrary lambda term.
Application	(MN)	A call to the lambda term M with the lambda term N as argument.

[72]Naturally, generic loops and conditionals could be added by a Python library.

24 Integrating Python Libraries via the Stub Language

So far, we did not explain, how the pipeline language (Chapter 23) knows which API elements are provided by the Safe-DS library or how third parties can integrate other libraries. Without this information, we cannot offer any static checks for Safe-DS pipelines.

For this purpose, we developed the *stub language*, which accurately captures the API of Python libraries but contains no implementation, akin to header files in the C language. We call the specification of a single API element *stub*. Unlike the stubs in MetaR [154], Safe-DS stubs include *type annotations* for parameters and results of functions. The typed stub language is an essential technical solution to support statically checked, embedded DSLs inside the pipeline language, e.g. for DS. The design of the stub language raises the following issues:

- **Describing the API for the pipeline language**: The stub language must specify which API elements may be used in the pipeline language, such as which functions may be called. We explain the concepts and syntax of the stub language in Section 24.1.
- **Safety**: The stub language must contain all information that is required to statically check programs written in the pipeline language, like the aforementioned type annotations for parameters and results of functions. Besides conventional type errors, we also want to catch violations of other preconditions (we call them *constraints*), like whether an argument for a numeric parameter is outside the legal range. The type and constraint system affect both the stub language and the pipeline language, and are discussed later in Chapter 25.
- **Connecting stubs and Python libraries**: The stubs contain no implementations, which are instead provided by Python libraries. To run Safe-DS pipelines (Part IV), we need to know how to map usages of stubs in the pipeline language to API elements of some Python library. This is explained in Section 24.2.
- **Documentation**: Library functions are only used if developers know about them, understand their semantics, and how to call them. Therefore, good documentation must be available, which requires a standardized format to document stubs, and, ideally, automatic generation of a self-contained API reference. This will be covered in Section 24.3.
- **Automated stub generation**: To make the integration of large libraries practical, stubs should be *generated automatically*. All the required information, like types and documentation, must be extracted from the source code of the integrated libraries. We address this issue in Section 24.4.

24.1 Concepts and Syntax

Core file structure. Listing 12 shows the contents of a simple stub file. It first declares its *package* (line 1), which is the namespace for stubs contained within. Names of global stubs[73] in the same package must be unique, but the same name may be used for global stubs in different packages. Global stubs can only be used in the pipeline language after importing them (Section 23.1), which requires the specification of the package to import from. Thus, packages prevent name collisions between global stubs with the same name that were integrated independently by different developers. Multiple stub files can belong to the same package.

Afterward, global stubs from other files are *imported* (line 3), so they can be referenced in this stub file. Imports of global stubs from the same package or a predefined `safeds.*` package may be omitted. The remainder of a stub file (lines 5–7) contains the declarations for stubs, as described below.

Classes. A *class* may contain *members*, which are attributes, methods, and nested classes. *Attributes* define data, while *methods* define behavior. *Nested classes* are classes that use the outer class as

[73] These are global classes and global functions. Stubs may also be nested inside another stub, e.g. a method of a class.

```
1 package safeds.data.tabular.containers
2
3 from safeds.data.image.containers import Image
4
5 class Table {
6     // See next listing
7 }
```

Listing 12: A stub file states its *package* (line 1), its imports (line 3) and finally a list of API element descriptions. Lines 5–7 show the start of a class description that is continued in Listing 13.

```
 5 class Table(data: Map) {
 6     attr rowCount: Int
 7
 8     static fun fromCsvFile(
 9         path: String,
10         separator: String = ","
11     ) -> table: Table
12 }
```

Listing 13: A class with a constructor (line 5), an instance attribute (line 6), and a static method (lines 8–11) with a required parameter (line 9), an optional parameter (line 10), and a single result (line 11). Attributes, parameters, and results have manifest types.

a namespace, so they do not pollute the global namespace. They may have members of their own. Attributes and methods may either be tied to the class itself (*static members*), or belong to a single instance of the class (*instance members*). Nested classes are always static members. Moreover, a class may have a *constructor* to create instances of it. Classes without a constructor may still be instantiated by other functions, e.g. by static factory methods [24].

The stub language can concisely describe the interface of a class, as demonstrated in Listing 13. The class `Table` has a constructor (line 5) that contains a list of parameters, here a single parameter `data` of type `Map`. No results are listed, as constructors are implied to return an instance of the class they belong to.

The members of the class are listed in its *body* between curly braces, here an instance attribute `rowCount` of type `Int` (line 6), and a static method `fromCsvFile` (lines 8-11) with a required parameter `path` of type `String` and an optional parameter `separator` of type `String` with the default value `","`. Note that attributes, parameters, and results have *manifest types* in the stub language, i.e. the types are explicitly specified rather than inferred.

In the pipeline language, calling a class invokes its constructor to create an instance. If the class has no constructor, it may not be called either. The call `Table({"a": [1, 2]})`, for example, would create a new `Table` instance. The behavior of the constructor (and any other stub) is not specified in the stub language but by the accompanying Python library (Section 24.2). Here the intention is to create a `Table` with one column (name `a`), and two rows (values `1` and `2`).

Members of a class are accessed by a member access (dot). Static members are accessed on the class itself, e.g. `Table.fromCsvFile("titanic.csv")`. Instance members are accessed on instances of the class, e.g. `Table({"a": [1, 2]}).rowCount`. Methods must be called immediately, as function pointers are not supported to offer a cleaner graphical view (Section 39.2). When calling a method, arguments for required parameters must be passed, while arguments for optional parameters may be omitted, in which case the default value is used. Arguments may be passed by position or name (Section 23.1).

```
// ...

fun parseInt(string: String) -> int: Int
```

Listing 14: A global function with a required parameter of type `String` and a result of type `Int`.

Global functions. Global functions can be thought of as static methods that reside outside a class in the global namespace. Listing 14 shows an example with a required `String` parameter and an `Int` result. As for methods, the parameters and results of global functions have manifest types.

Global functions must be imported before use in the pipeline language. Calling them runs the code in the associated Python function (Section 24.2). The Safe-DS library does not have global functions (Section 16.1.1), but other Python libraries offer them (Chapter 14).

Enumerations. Finally, we want to offer a concise way to avoid parameter dependencies in stubs (Section 5.2), where the behavior of a parameter depends on the value of another. The constructor for a support vector machine in scikit-learn, for instance, has the parameters `kernel`, which defines the kernel to use (`"linear"`/`"poly"`/...), and `degree` to set the degree of a polynomial kernel. Naturally, setting the degree only makes sense if a polynomial kernel is used. The API design does not highlight this restriction, though, and also allows passing the degree for linear kernels.

Enumerations are containers for structured data without behavior, inspired by Rust [161]. They contain a fixed, finite set of *variants*, which all have their own constructor. Different variants of the same enumeration may have parameter lists with different lengths, and accept parameters with different names and types. Each constructor parameter automatically defines an attribute with the same name and type for the variant, so the argument passed to a parameter can be accessed later. Enumerations may be global or nested in a class.

Listing 15 shows an enumeration named `Kernel` that describes the possible kernels of a support vector machine. It has two variants: `Linear` accepts no parameters (and has no attributes)[74], and `Polynomial` takes a parameter `degree` with type `Int` (and has a corresponding attribute).

In the pipeline language, variants of an enumeration can be instantiated by a member access and subsequent call, e.g. `Kernel.Linear()`[75] or `Kernel.Polynomial(3)`. As usual, arguments may be passed by position or name. Attributes can be accessed by another member access on a variant instance (e.g. `Kernel.Polynomial(3).degree`). Enumerations make it impossible to accidentally set or access the `degree` for an instance of the `Linear` variant.

```
// ...

enum Kernel {
    Linear()
    Polynomial(degree: Int)
}
```

Listing 15: An *enumeration* defines a finite, fixed set of *variants*. Each variant has its own list of parameters. Each parameter automatically defines an attribute for the variant with the same name and type.

[74]For parameterless variants, the parentheses for the constructor may be omitted.
[75]For parameterless variants, the parentheses for the call may be omitted.

24.2 Mapping Stubs to Python

Stubs fulfill two main purposes: First, they describe the API elements that may be used inside the pipeline language (Section 24.1). Second, they describe which API elements of a Python library implement the corresponding functionality. The latter is the topic of this section.

Default mapping of global classes and functions. By default, a global class or function with the name `x` in the Safe-DS package `y` is mapped to the Python declaration with the name `x` in the Python module `y`. The class `Table` in Listing 12, for example, corresponds to the Python declaration that can be reached with the Python import `from safeds.data.tabular.containers import Table`.

A Safe-DS class must correspond to a Python class, and a Safe-DS function must correspond to a Python function, which must be able to handle all legal calls of the Safe-DS function. The constructor of a Safe-DS class corresponds to the `__init__` method of the associated Python class, which again must be able to handle all legal invocations of the Safe-DS constructor.

Note that all stubs must correspond to some API element of a Python library, but that it is unnecessary to create stubs for all API elements of a Python library. Thus, the stub language can act as a facade [24].

Default mapping of class members. Assuming that we already connected the contained Safe-DS and Python classes, their members are again mapped by name: A Safe-DS class member with the name `x` corresponds to the Python class member with the name `x` by default. The kinds of class members must match:

- A Safe-DS instance attribute must correspond to a Python instance attribute or property[76].
- A Safe-DS static attribute must correspond to a Python class attribute.
- A Safe-DS instance method must correspond to a Python instance method, which must be able to handle all legal calls of the Safe-DS method.
- A Safe-DS static method must correspond to a Python method with the `@staticmethod` or `@classmethod` decorator [162], which must be able to handle all legal calls of the Safe-DS method.
- A Safe-DS nested class must correspond to a Python nested class. Naturally, their members must match, too.

Listing 16 shows a possible Python class for the Safe-DS class from Listing 13. The actual implementation has been replaced with ... for brevity.

Default mapping of enumerations. Safe-DS enumerations are either global or nested in a class, so they are mapped according to the naming rules outlined above, too. However, enumerations do not have a direct equivalent in Python and are instead modeled by nested classes. The enumeration

```python
class Table:
    def __init__(data):
        ...

    @property
    def rowCount():
        ...

    @staticmethod
    def fromCsvFile(path, separator=","):
        ...
```

Listing 16: A Python class that corresponds to the Safe-DS class from Listing 13. For brevity, the implementations of the methods are omitted.

[76] This is a method with the `@property` decorator [162].

```python
class Kernel(ABC):
    class Linear:
        pass

    class Polynomial:
        def __init__(degree):
            self.degree = degree
```

Listing 17: Python classes that correspond to the Safe-DS enumeration from Listing 15.

corresponds to an abstract outer class and each variant to an inner class, which each have a suitable constructor that defines parameters and attributes. By default, variants are mapped to Python classes by name. Listing 17 shows the Python classes for the enumeration from Listing 15.

As we will see in Section 25.2, the subtyping rules for enumerations are stricter than could be expressed with either Safe-DS or Python classes, which enables more thorough static analysis. Because of this, we did not simply use nested classes in Safe-DS, too.

Renaming API elements. Listing 13 shows the stub for a method `Table.fromCsvFile` method (camel case), yet the Safe-DS library defines the method `Table.from_csv_file` (snake case). This does not match the default mapping rules outlined above and would lead to errors when executing a pipeline that calls this method (Part IV).

Such discrepancies are possible due to the `@PythonName` annotation. An *annotation* adds metadata to a stub. The `@PythonName` annotation defines the name of the corresponding API element in the Python library, which must otherwise be identical to the name of the stub. If this annotation is used, the name of a stub only specifies how it must be used in the pipeline language.

In Listing 18, the `Table` class in the Python library has an attribute called `row_count` and a static method called `from_csv_file`, but they can be used under the names `rowCount` and `fromCsvFile` in the pipeline language respectively.

Hence, the `@PythonName` annotation provides support for rename refactorings [11], without needing to alter the API of the library itself (Part I). For us, this is useful, so the Safe-DS library can adhere to the name conventions in Python, but be used under our name convention in the pipeline language[77]. For third parties, it is also useful to clarify names as needed.

Moving API elements. Similarly, the `@PythonModule` annotation can move global API elements [11], e.g. to combine declarations from various Python modules in the same Safe-DS package. It is added above the package declaration of a stub file and specifies the Python module that contains the imple-

```
class Table {
    @PythonName("row_count")
    attr rowCount: Int

    @PythonName("from_csv_file")
    static fun fromCsvFile(
        path: String,
        separator: String = ","
    ) -> table: Table
}
```

Listing 18: The `@PythonName` annotation allows using API elements of a Python library under a different name in the pipeline language.

[77] We picked camel case instead of snake case, since it is more compact and easier to type.

```
1 @PythonModule("left_pad")
2
3 package utils
4
5 fun leftPad(string: String, length: Int)
```

Listing 19: The `@PythonModule` annotation decouples the structure of Safe-DS packages from Python modules. It affects all stubs in the file.

mentation for the stubs in the file. This module must otherwise be equivalent to the Safe-DS package, as explained above. Within the pipeline language, only the Safe-DS package matters.

Listing 19 illustrates the use of the `@PythonModule` annotation. In the pipeline language, the stub `leftPad` may be imported from the package `utils`, but its implementation resides in the Python module `left_pad`.

Macros. Macros are a last resort if a simple rename is not sufficient. They are intended to define methods for core types, like strings[78], and allow mapping calls in the pipeline language to an arbitrary Python expression. They are not meant for large design changes to a library, which should be done using adaptoring instead (Part I).

For example, we want to be able to access the length of a string by calling `"Hello, world!".length()` in the pipeline language, for the reasons outlined in Section 16.1.1. In Python, however, this would need to be written as `len("Hello, world!")`.

Listing 20 shows how this can be accomplished: A method stub (line 3) defines the interface for the pipeline language. Additionally, a *template* passed to the `@PythonMacro` annotation defines the corresponding Python expression (line 2). The template can contain *template expressions*, which either refer to the receiver of the call (written as `$this`), or some parameter `param` of the annotated method (written as `$param` and not shown here).

24.3 Documentation

Structured documentation comments. The stub language also defines a single format for documentation comments, which is inspired by JavaDoc [163]. An example for the method `Table.fromCsvFile` is shown in Listing 21. First, the comment includes a description of the method itself (line 4). *Tags* segment the rest of the comment:

- The `@param` tag adds a description for a parameter of the method. The name of the documented parameter is written after the tag.
- The `@result` tag does the same for a result of the method. Association with the result again happens by name, which is one of the reasons that results are named.
- The `@example` tag is used to show how the method should be used. The tag can be repeated to include multiple examples.

```
1 class String {
2     @PythonMacro("len($this)")
3     fun length() -> length: Int
4 }
```

Listing 20: With the `@PythonMacro` annotation, calling a function in the Safe-DS language can run an arbitrary Python expression.

[78] We do not want to bloat the Safe-DS library with adapter classes for core types.

```
1     // ...
2
3     /**
4      * Create a table from a CSV file.
5      *
6      * @param path
7      * The path to the CSV file.
8      * @param separator
9      * The separator between the values in the CSV file.
10     *
11     * @result table
12     * The created table.
13     *
14     * @example
15     * pipeline example {
16     *     val result = Table.fromCsvFile("input.csv");
17     * }
18     */
19    static fun fromCsvFile(
20        path: String,
21        separator: String = ","
22    ) -> table: Table
23 }
```

Listing 21: Documenting declarations in the stub language using a JavaDoc-like format. The comment starts with a description of the documented method (line 4). Its parameters and results are then documented individually by using the `@param` and `@result` tags, followed by the name of the documented parameter and result. Examples are added using the `@example` tag.

All natural-language descriptions can include Markdown [164] for formatting, e.g. to italicize text or add tables. Finally, links to other API elements can be added via a `@link` inline tag: For instance, `{@link Table.fromCsvFile}` would point to the method `Table.fromCsvFile`.

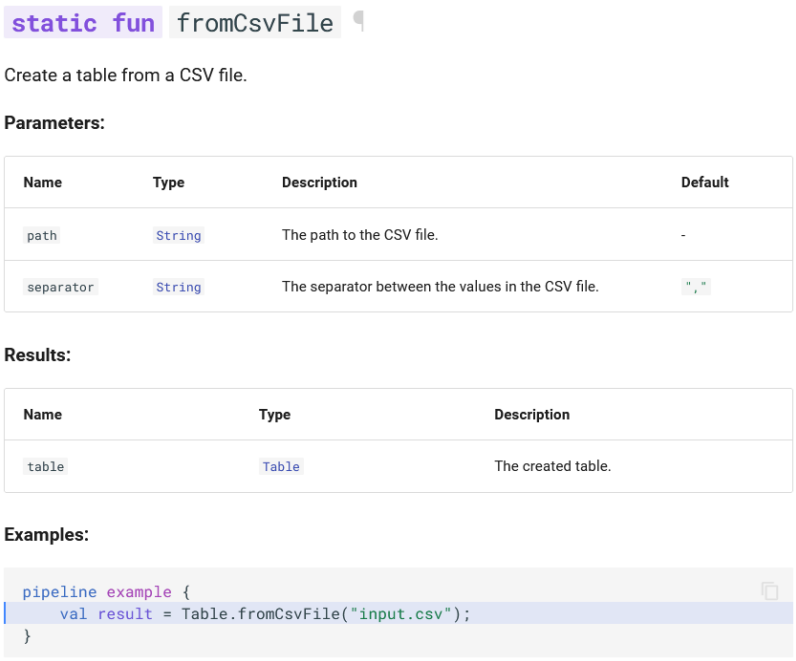
Specifying maturity of API elements. Furthermore, the maturity of an API element can be described via annotations: An API element can be marked as `@Experimental` to indicate that it might be changed or removed in the future without deprecation cycles.

It can also be marked as `@Deprecated` to indicate that it should no longer be used. A reason for deprecation, a possible alternative, and a version when the API element will be removed can be specified by passing arguments to the `@Deprecated` annotation. Listing 22 shows an example for this.

Automated generation of an API reference. Stubs tell the pipeline language, which API elements are available and how to use them. But users of the language must also know this. Luckily, stubs provide all the information for the automatic generation of a polished API reference, similar to the `javadoc` tool [163] for Java. Figure 29 shows the result for the `Table.fromCsvFile` method from Listing 21.

```
5  @Deprecated(
6      alternative="Table.fromCsvFile",
7      reason="We prefer methods to global functions.",
8      removalVersion="0.31.0"
9  )
10 fun readCsvFile(path: String) -> table: Table
```

Listing 22: The `@Deprecated` annotation stipulates that an API element should no longer be used and can suggest alternatives.



`static fun` `fromCsvFile`

Create a table from a CSV file.

Parameters:

Name	Type	Description	Default
`path`	`String`	The path to the CSV file.	-
`separator`	`String`	The separator between the values in the CSV file.	`","`

Results:

Name	Type	Description
`table`	`Table`	The created table.

Examples:

```
pipeline example {
    val result = Table.fromCsvFile("input.csv");
}
```

Figure 29: Automatically generated API reference for the `Table.fromCsvFile` method from Listing 21. Types are clickable and jump to the corresponding part of the API reference. Usages of the documented API element in examples are highlighted.

We combine natural-language descriptions and examples from documentation comments with type annotations of parameters and results in the stub code. The optionality and potential default value of parameters is extracted, too. Types, such as `String`, are hyperlinks that jump to the corresponding section of the API reference. In examples, all usages of the documented API element are highlighted, so they can easily be identified even in longer examples.

The generator also takes maturity annotations into account. Experimental declarations are marked with the icon 🧪. Deprecated declarations are indicated by a ⚠️ icon, and all additional information, like the alternative to use, is included above the description of the API element.

The API reference for API elements that are built into the Safe-DS language is hosted online, e.g. for the `Table` class[79].

24.4 Automated Stub Generation

To easily create correct stubs even for large libraries, automation is crucial. Luckily, the code and docstrings (Listing 1) of Python libraries already contain parts of the information that is needed to create stubs:

- Names of packages *must be* defined by the module structure of the library. However, there are typically multiple paths to a single declaration of a Python library[80]: First, via the module that includes the declaration. Second, via `__init__.py` modules that bundle and reexport declarations from multiple modules [165]. Hence, the package to choose in the stub language can be ambiguous.
- Names of classes, attributes, methods, parameters, and global functions *must be* part of the code.
- Names of results *may be* included in the docstring.
- Types of attributes, parameters, and results *may be* manifested in the code by type hints [28].

[79] https://dsl.safeds.com/en/stable/api/safeds/data/tabular/containers/Table

[80] In other words, there are multiple legal ways to import the same declaration in a client program written in Python.

- Types of attributes, parameters, and results *may be* inside docstrings, albeit in a less standardized format than type hints. They might also conflict with the type hints if both are provided.
- Descriptions of all kinds of API elements *may be* part of the docstring.

Thus, initial stubs can be generated automatically[81], drastically reducing the time required to integrate libraries. Manual augmentation of the result is only needed if information is missing from the library. We use the following heuristics to resolve problems during stub generation:

- If multiple packages are valid, we pick the shortest one. Client programs in Python typically do the same.
- If names of results are missing, results are called `result1`, `result2`, etc. unless such a name collides with a result that has been named manually. In this case, the next free number is used.
- If types can neither be extracted from type hints nor from a docstring, the special `unknown` type is used in the stub and a TODO comment is added to alert the user that they must manually specify the type.
- If type information from a type hint collides with the one extracted from a docstring, we use the information from the type hint and warn by default. This expresses that we trust type hints more, since type hints are more formalized, and we assume that library developers rather forgot to update the docstring. Users of our tool can instead decide to use types from docstrings, disable the warning, or raise an error.
- If the description of an API element is missing in the library's docstring, the corresponding tag (or the entire documentation comment) is omitted in the stub.

Users can also optionally let the generator add `@PythonName` annotations, so names adhere to the name convention of the Safe-DS language. An annotation is only added if the Safe-DS name and the Python name differ.

We use the stub generator to integrate the Safe-DS library into the Safe-DS language, which offers the required expressiveness. Releases of the Safe-DS language contain the created stubs. Third parties can employ the generator to extend the language with new libraries.

25 Safety

One of our goals (Chapter 22) is to detect misuse of the integrated libraries early to prevent expensive errors late into a pipeline run. In particular, we focus on the following broad categories of errors:

- Conventional type errors (e.g. passing a `String` to a parameter of type `Int`)
- Boundary errors (e.g. passing the value `-2` to a parameter that denotes the maximum depth of a decision tree)
- State errors (e.g. using an untrained model for inference)
- Schema errors[82] (e.g. accessing a column that does not exist in a table)

Type errors are caught *statically* without running the pipeline at all. This *type checking* validates whether some actual type (e.g. `String`) is compatible to an expected type (e.g. `Int`) for some fixed language concepts. In a call, for instance, the actual type comes from an argument, whereas the expected type comes from the corresponding parameter. Before we can tackle type checking (Section 25.4), we must discuss several prerequisites, though:

- Types (Section 25.1): We need to somehow express the actual and expected types.
- Subtyping (Section 25.2): We need to know which types are compatible to each other.

[81] This is similar to the generation of adapters in Part I. However, stubs are meant to describe the status quo of an API (except for renamings and moves), while adaptoring intends to fundamentally change it. It is possible to combine both, by first using adaptoring to create an adapter library, and then generating stubs to integrate it into the language.

[82] Schema errors are specific to the DS domain. The other three errors might occur when developing arbitrary pipelines.

- Type inference (Section 25.3): In previous chapters, we have noted that parameters and results of segments in the pipeline language, as well as parameters, results, and attributes in the stub language have *manifest types*, i.e. they are explicitly specified by developers in the code with *type annotations*. The types for all other language concepts are not contained in the code (usability goal, Chapter 22), so they must be *inferred*.

Checks for boundary, state, and schema errors are not part of the type system. Instead, these checks can be expressed by adding *constraints* to callables (Section 25.5). Constraints can only express preconditions of callable, whereas types may express preconditions (parameter types) and postconditions (result types)[83]. Boundary and state errors are caught statically, while the detection of schema errors requires partial execution of a pipeline.

25.1 Types

The type system needs to cope with the needs of the pipeline and the stub language: We want type checking (Section 25.4) in the pipeline language to guarantee the validity of the operations that are performed in it. These operations are defined by assignments and expressions, which have no manifest types. Thus, we must infer the types of placeholders and any expression (Section 25.3), and need concepts to denote the result internally.

In the stub language, we must accurately describe the types of parameters, results, and attributes of an external Python library using manifest types[84]. Generally, parameter types are required for type checking in the pipeline language, whereas result and attribute types are needed for type inference.

To accurately describe *any* Python library with our type system, we would have to offer an equivalent to any Python type hint [28]. This would also include types for Python concepts, like `**kwargs` parameters [136], that we deliberately avoided when designing the Safe-DS library (Section 16.1.2).

We decided to include only types initially that are needed to integrate the Safe-DS library. The offered types are still sufficient for most API elements of the third-party libraries we investigated in Chapter 14, and new types can be added in the future as required. The resulting type system is overall heavily inspired by Kotlin [167].

Section 25.1.1 explains the types that are needed to safely integrate the Safe-DS library. Then, Section 25.1.2 discusses which types must be added for type inference in the pipeline language.

25.1.1 Types for Library Integration

Class types. All data is represented by objects, i.e. instances of some class, in Python [134]. Hence, restricting the legal values of a parameter to instances of some class is the most essential operation. For example, the `path` parameter of the method `Table.from_csv_file` must be a string.

Each class in the stub language defines a new *class type*, which contains all instances of the class. The corresponding type annotation simply includes the name of the class, e.g. `String` accepts any string.

`String` is a class type that is built into the language among a few others like `Boolean`, `Float`, and `Int`, rather than provided by the Safe-DS library, as Python already defines the corresponding `str`, `bool`, `float`, and `int` classes itself. It is still described by a normal class stub and treated like any other class type regarding subtyping (Section 25.2).

Enumeration types. We introduced enumerations (Section 24.1) as a solution for parameter dependencies (Section 5.1), where the value of a parameter alters the semantics of another. Listing 15 shows

[83] Boundary checks could also be described by *refinement types* [166], which narrow a type by some predicate that must hold for all values. For us, expressing boundaries in preconditions suffices, though.

[84] Types of parameters and results of segments are specified in the same manner, so we will not discuss this further.

an enumeration that models the possible kernels of a support vector machine. We still need to restrict the values that the `kernel` parameter of the constructor for a support vector machine accepts, though.

This is done by annotating the parameter with an *enumeration type*, which consists of the name of an enumeration and only allows instances of variants of this enumeration. For example, annotating the `kernel` parameter with the type `Kernel` would only allow passing instances of the `Linear` or `Polynomial` variants, as desired.

Variant types. To model state constraints (Section 25.5), each variant of an enumeration also defines its own *variant type*, which only accepts instances of that variant. For example, a parameter with the manifest type `Kernel.Polynomial` would only allow instances of the `Polynomial` variant.

Explicit nullability. We subsume class types, enumeration types, and variant types by the term *named types*. By default, the `null` literal, which is equivalent to Python's `None`, cannot be assigned to a parameter with a named type. This prevents null pointer exceptions when accessing members of the parameter, either in the code of a segment or in the body of the corresponding Python function.

Appending a question mark to a named type, denotes the type as *nullable*, meaning it accepts `null` in addition to the original values. For example, the type `Table?`[85] allows any instance of the `Table` class and the `null` literal.

When accessing members of a nullable type in the pipeline language, a special *null-safe* member access (`?.`) must be used, for example `table?.rowCount`. If `table` is `null`, the entire expression evaluates to `null`. Otherwise, the `rowCount` is computed normally. As with any type annotation, parameters of stubs must only be marked as nullable, if the corresponding Python code can handle `None`.

Literal types. In Section 5.1 we discussed literal sets, where a parameter may only accept a finite set of values. For instance, plot methods in the Safe-DS library have a `theme` parameter to switch between a light and dark mode (Section 16.5), which only accepts the values `"light"` and `"dark"`.

Such cases can be described exactly by *literal types*, which explicitly denote the non-empty, finite set of values they allow. The `theme` parameter has the type annotation `literal<"light", "dark">`, and accepts only the two listed strings. Using the class type `String` instead, would incorrectly allow passing other strings.

Callable types. The Safe-DS library has higher-order functions like `Table.remove_rows`, which expect callbacks. In the pipeline language, such functions are called with a lambda. However, we cannot simply pass any lambda, but it must have a suitable signature (more on subtyping rules for callable types in Section 25.2).

The expected signature of callbacks can be annotated by *callable types*. A callable type lists the names and types of its parameters and results. `() -> ()` is the most basic callable type without parameters and results. `(row: Row) -> (doRemove: Cell)` is a callable type with one parameter named `row` with class type `Row`, and one result called `doRemove` with class type `Cell`.

Names are only included for documentation, so users know the role of parameters and results, particularly if there are multiple. Parameter types are used to infer the types of lambda parameters (Section 25.3), while result types are used for type checking (Section 25.4).

Union types. Python does not support function overloading[86], so function parameters often accept instances of several unrelated classes. For example, the method `Regressor.predict` of the Safe-DS library may be called with either a `Table` or a `TabularDataset`.

[85] Read this as "Table or null".

[86] The `@overload` decorator is only used for typing and has no effect at runtime [28].

These cases can be captured exactly using *union types*, which have a non-empty, finite set of type options and accept a value if at least one of the options accepts it. For example, the type `union<Table, TabularDataset>` allows all instances of the `Table` class, and all instances of the `TabularDataset` class.

Working with union types can be difficult, as clients must only interact with the interface that is shared by all options. Accessing members that are only available on some options requires type casts. Hence, mypy suggests avoiding union types as return types of functions [168], advice heeded by the Safe-DS library. Generalized for the stub language, union types must not be used in out-positions (see Section 25.2), so users of the pipeline language must never interact with expressions that have a union type.

Type parameters. Say, we assign a list literal to a placeholder: `val list = [1, 2, 3];`. What is the type of the expression `list.at(0)`, where `at` is supposed to return the element at index `0`? Clearly, `list` holds an instance of the class `List`, and its first element is `1`, so `Int` would be the answer.

But to infer the type of the expression `list.at(0)` in general (Section 25.3), knowing that `list` has type `List` is not enough, since we may not know the exact values that are contained in the list. `list` may also be a parameter, for instance. We must instead know that `list` is a `List` *that contains elements of type* `Int`.

We could define a class `IntList` with the required semantics for its `at` method:

```
class IntList {
    fun at(index: Int) -> element: Int
}
```

But then we would (1) need to define other classes for `StringList` etc., and (2) tell the type inferrer that the list literal `[1, 2, 3]` should have type `IntList`. Since lists may be nested arbitrarily, this clearly does not work. We would need an infinite number of classes to cover all cases.

Instead, we define `List` only once as a *generic* class, which is a class with type parameters. *Type parameters* are declared after the name of the class and may be used in type annotations within the body of the class:

```
class List<E> {
    fun at(index: Int) -> element: E
}
```

Type parameters may be optional and have a default type (e.g. `class List<E = Int>`). Whenever a class type with type parameters is used as a manifest type, required type parameters must be and optional type parameters may be substituted by *type arguments*, e.g. `List<Int>`. Type arguments are passed to type parameters by position or name, similar to arguments of calls. Type arguments are inferred when calling the constructor of a class (or in this special case when using a list literal).

In either case, all occurrences of the type parameter in type annotations within the class are replaced by the type argument. Here, the `at` method would correctly express that it returns an `Int`.

Type parameter bounds. Type parameters can optionally have some type as an upper bound. The manifest or inferred type argument must be a subtype of the upper bound (Section 25.2). For example, the type parameter `E` of a `class MyList<E sub Number>` would only accept types that are a subtype of the class type `Number` as type argument. We are going to use this for state constraints (Section 25.5).

25.1.2 Types for Intermediate Inference Results

The type system also includes two kinds of types that occur as intermediate results during type inference (Section 25.3) but cannot be used in type annotations. These are required, so we can denote the type of any expression in the pipeline language.

Static types. The constructor call `Table()` instantiates the `Table` class, so it has the class type `Table`. However, we must also be able to express the type of the reference `Table`. *Static types* cover this case, which are the types of references to a class. Calling an expression with a static type invokes the constructor, provided that it exists. Furthermore, static members may be accessed on an expression with a static type, hence the name.

Named tuple types. Callables in the Safe-DS language may return multiple or no results. We need to be able to denote the types of calls to them. This is done with *named tuple types*, which consist of a list of names and types of results. They correspond to the part after the arrow in callable types. For example, a call to an expression with type `() -> ()` would return an expression with an empty named tuple type.

25.2 Subtype Relation

For type checking, we need to know which types are compatible with each other, i.e. whether a value of type A may be used if a value of type B is expected. This is the case, if the operations that are legal on values of type B are also legal for all values of type A. This is trivially true, if A and B are identical, but this notion is too restrictive: For example, passing an integer argument to a number parameter is perfectly acceptable.

> **Definition 6** (Subtype/supertype): Let $\mathbb{T}$ denote the set of valid types for a given Safe-DS program. If all values of type $A \in \mathbb{T}$ support all operations that are legal for the values of type $B \in \mathbb{T}$, we say that
>
> - A is a *subtype* of B[87] (written as $A <: B$), or
> - B is a *supertype* of A (written as $B :> A$).

$\mathbb{T}$ depends on the class, enumeration, and variant declarations, which all introduce new named types (Section 25.1)[88]. The relations in Definition 6 have the following properties for all $A, B, C \in \mathbb{T}$ [169]:

- $A <: A$ (reflexivity)
- $(A <: B) \land (B <: C) \Rightarrow A <: C$ (transitivity)
- $A <: B \Leftrightarrow B :> A$

In this section, we explain the subtype relation between all kinds of types in the language (Section 25.1). We omit the discussion of subtyping for intermittent types for type inference (Section 25.1.2), since they never occur in a position where subtyping matters inside legal programs[89].

Common super- and subtype. When inferring type arguments (Section 25.3), we must determine the lowest common supertype or highest common subtype of a list of types. Here, having a guaranteed common supertype and subtype is beneficial. Hence, the type system has two special class types, `Any` and `Nothing` with fixed subtyping rules:

- `Any` is a supertype of any *non-nullable* type, which are types that do not accept `null`.
- `Any?` also allows `null` and is, thus, a supertype of any type.
- `Nothing` is a subtype of any types.

`Any` defines a few general methods, like `toString` that are available on all values. The type `Nothing` is empty, i.e. no value in the language has this type. `Nothing?` is the type that contains only `null`.

[87] If we intuitively regard types as a set of values, a subtype relation corresponds to set inclusion [169].

[88] A full formalization of this notion requires a complete specification of the syntax of the language, e.g. using the extended Backus-Naur form [170], and a complete specification of the semantics of the language, e.g. using structural operational semantics [171]. This is out of the scope of this work, since the language and type concepts we use are already well-researched, e.g. in [172].

[89] The implementation (Chapter 26) still covers those types to reduce the number of errors shown to users.

Table 14: Is the non-nullable type A (rows) a subtype of the non-nullable type B (columns)? Cases with a ✓ are always true, cases with a × are always false, and cases with a ? are discussed further in this section. "Other class" stands for any class type but `Nothing` and `Any`.

	`Nothing`	`Any`	Other class	Enum	Variant	Literal	Callable	Union
`Nothing`	✓	✓	✓	✓	✓	✓	✓	✓
`Any`	×	✓	×	×	×	×	×	?
Other class	×	✓	?	×	×	×	×	?
Enum	×	✓	×	?	×	×	×	?
Variant	×	✓	×	?	?	×	×	?
Literal	×	✓	?	×	×	?	×	?
Callable	×	✓	×	×	×	×	?	?
Union	?	✓	?	?	?	?	?	?

Simple cases. For some types $A, B \in \mathbb{T}$, we can quickly determine whether $A <: B$:

1. If A is nullable, but B is non-nullable, then $A \not<: B$.
2. Otherwise, let A' and B' be the non-nullable versions of A and B. Then $A <: B \Leftrightarrow A' <: B'$.

Based on the kind of A' and B', we can often quickly decide whether $A' <: B'$. This is summarized in Table 14, where the rows correspond to A' and the columns to B'. ✓ indicates $A' <: B'$, and × indicates $A' \not<: B'$. The remaining cases with a ? are covered below.

Literal types. We first explain the subtyping rules for the "Literal" row in Table 14: Let L_1 be a literal type with the value options[90] $\mathbb{V}_1$, and L_2 be a literal type with the value options $\mathbb{V}_2$. Then

$$L_1 <: L_2 \Leftrightarrow \mathbb{V}_1 \subseteq \mathbb{V}_2 \text{ (literal/literal)}$$

Let C be any class type, and U_1 be the union type with the type options[91] that are created by replacing the value options of L_1 with class types as described in Table 15. Then

$$L_1 <: C \Leftrightarrow U_1 <: C \text{ (literal/class)}$$

Let U be a union type with the type options $\mathbb{O}$. Then

$$L_1 <: U \Leftrightarrow \forall v \in \mathbb{V}_1 : \exists O \in \mathbb{O} : \texttt{literal<}v\texttt{>} <: O \text{ (literal/union)}$$

Here, `literal<`v`>` is a literal type with a single value option.

Table 15: Converting value options of a literal type to a class type.

Literal Kind	Example	Class Type
Boolean	`true`	`Boolean`
Floating point	`1.2`	`Float`
Integer	`1`	`Int`
String	`"foo"`	`String`

[90] For example, `literal<1, "foo">` would have the options `1` and `"foo"`.

[91] For example, `union<Int, String>` would have the options `Int` and `String`, which are both class types.

Union types. We now cover the subtyping rules for the remaining entries in the "Union" row and "Union" column of Table 14: Let U be a union type with the type options $\mathbb{O}$, B be any type, and A be any type that is neither a union nor literal type. Then

$$U <: B \Leftrightarrow \forall O \in \mathbb{O} : O <: B \text{ (union/other)}$$
$$A <: U \Leftrightarrow \exists O \in \mathbb{O} : A <: O \text{ (other/union)}$$

Enumeration types and variant types. Next, we discuss the entries for enumeration and variant types in Table 14: Let E_1 and E_2 be enumeration types, V_1 and V_2 be variant types, and d be the function that returns the declaration that a named type is derived from. Then

$$E_1 <: E_2 \Leftrightarrow E_1 = E_2 \quad \text{(enum/enum)}$$
$$V_1 <: V_2 \Leftrightarrow V_1 = V_2 \quad \text{(variant/variant)}$$
$$V_1 <: E_1 \Leftrightarrow d(V_1) \text{ is variant of } d(E_1) \quad \text{(variant/enum)}$$

Basic subclassing. Subtype relations between class types other than `Nothing` or `Any` are defined *nominally*, i.e. a class stub explicitly specifies its superclass. For instance, a `Regressor` is a subtype of `SupervisedModel`, which is written as `class Regressor sub SupervisedModel`. The superclass need not know its subclasses, so arbitrary subclasses may be added. In contrast, the variants of an enumeration are fixed. Our type system only supports singular inheritance because the public, high-level abstractions provided by the Safe-DS library form a tree-like hierarchy[92].

Let C_1 and C_2 be class types other than `Nothing` and `Any` without type arguments, and d be the function that returns the declaration that a named type is derived from. Then

$$C_1 <: C_2 \Leftrightarrow (C_1 = C_2) \lor (\exists C_3 : d(C_1) \texttt{ sub } C_3 \land C_3 <: C_2)$$
$$\text{(other class/other class, no type arguments)}$$

Class types with type arguments (example). We explain the subtyping rules for parametrized class types using the following stubs for a simple stack and a function that does unknown work on a stack:

```
class Stack<T> {
    fun push(value: T)
    fun pop() -> value: T
}

fun mystery(stack: Stack<Number>)
```

Can we assign a `Stack<Int>` to the parameter `stack`, i.e. is `Stack<Int>` a subtype of `Stack<Number>`? The answer is no because `mystery` could push a `Float`, which the caller of `mystery` might not be able to handle afterward. Can we instead assign a `Stack<Any>` to the parameter `stack`? Again, this is not possible, since `pop` may return values that are not `Numbers`, which `mystery` might not be able to handle.

If the class `Stack` only had a `pop` method, passing a `Stack<Int>` would be fine. After `mystery` is done, the stack would still only contain `Ints`. Likewise, if the class only had a `push` method, we could pass a `Stack<Any>`. `mystery` would restrict itself to only pushing `Numbers`, but work normally.

Variance. In general, given a generic class `Container<T>`, and types `A`, and `B`, where `A` is a subtype of `B`, the following rules apply [169]:

- `Container<A>` is a subtype of `Container<B>`, iff `T` only occurs in *out-positions*, like in type annotations of attributes and results. In this case, we say that `T` is *covariant*.

[92] The library does use multiple inheritance to eliminate code duplication, e.g. by sharing an implementation between a `DecisionTreeClassifier` and a `DecisionTreeRegressor` in addition to extending `Classifier` and `Regressor` respectively. But this is an internal implementation detail and, thus, irrelevant for stubs, which only describe an API.

- `Container<B>` is a subtype of `Container<A>`, iff `T` only occurs in *in-positions*, like in type annotations of parameters. In this case, we say that `T` is *contravariant*.
- If neither of the criteria are true, we say that `T` is *invariant*.
- If both criteria are true, we say that `T` is *bivariant*, but this means that `T` is never used.

Languages can either infer variance from how a type parameter is used or let the user specify variance explicitly and then check whether all usages of the type parameter are correct. We chose the latter approach because stubs might omit members of a Python class. Hence, the Python implementation of `Stack`, which is used internally by `mystery`, might have a `push` method, even if the stubs do not list it.

By default, type parameters are invariant. Adding the `out` modifier to a type parameter (e.g. `class Stack<out T>`) makes it covariant, and it must then only be used in out-positions. Likewise, the `in` modifier marks a type parameter as contravariant and usages are constrained to in-positions. Bivariant type parameters are not modeled because they can simply be omitted.

Callable types and nested parametrized types, like `Stack<Stack<T>>`, complicate the distinction between out-positions and in-positions. For example, parameter type annotations are normally in-positions. However, in the following code, the contravariant type parameter `T` in fact occurs in an out-position in line 3, which is illegal[93]:

```
1 class Container<in T> {
2     fun print(
3       formatter: (value: T) -> text: String
4     )
5 }
```

We determine the position of a type parameter use by the following pseudocode algorithm:

1. Set `currentNode` to the AST node that represents the type parameter use.
2. Set `position` to `1`.
3. While `currentNode` is not the class that defines the type parameter:
   a) If `currentNode` occurs in Table 16, multiply `position` by the listed multiplier.
   b) Set `currentNode` to the container of `currentNode`.
4. Return `position`.

A result of `-1` indicates an in-position, `0` an invariant-position[94], and `1` an out-position. For the code above, we would correctly determine that `T` occurs in an out-position (`1`), since it is nested inside two parameter type annotations (multiplier `-1`), one for the parameter `value` of the callable type, and another for the parameter `formatter` of the method `print`.

Table 16: Places where a type parameter may be used and their base position. We assign a numeric multiplier to each kind of position (in = −1, invariant = 0, out = 1). The position of the type parameter can then be determined by traversing the AST and aggregating all multipliers.

Current node	Base position	Multiplier
Attribute type annotation	out	1
Parameter type annotation	in	−1
Result type annotation	out	1
Type argument (invariant type parameter)	invariant	0
Type argument (covariant type parameter)	out	1
Type argument (contravariant type parameter)	in	−1

[93] Consider whether a `Container<Any>` can be assigned to a `Container<Int>`, as implied by the contravariance.
[94] This denotes a position that is neither in- nor out-position and is only caused by invariant type parameters.

Class types with type arguments. We are now ready to specify the subtype relation for two arbitrary class types C_1 and C_2 other than `Nothing` and `Any`. We need the following definitions:

- d is the function that returns the declaration that a named type is derived from.
- T_{inv}, T_{co}, and T_{contra} are the sets of invariant, covariant, and contravariant type parameters of $d(C_2)$.
- σ_1 and σ_2 are the substitution maps of type parameters by type arguments in C_1 and C_2.
- $\sigma_i[t]$ is the type argument assigned to type parameter t in σ_i.
- C^σ is the type created by substituting in class type C all references to type parameters contained in σ by the associated type argument.

For example, for the `class Container<T>`, $T_{\text{inv}} = \{\texttt{T}\}$, and $T_{\text{co}} = T_{\text{contra}} = \emptyset$. The type `Container<Int>` has the substitutions $\{\texttt{T} \to \texttt{Int}\}$, and can be built from the type `Container<T>` with these substitutions.

The subtyping rule "other class/other class" is

$$
\begin{aligned}
C_1 <: C_2 \Leftrightarrow \big(& \\
& (d(C_1) = d(C_2)) \land \\
& (\forall t \in T_{\text{inv}} : \sigma_1[t] = \sigma_2[t]) \land \\
& (\forall t \in T_{\text{co}} : \sigma_1[t] <: \sigma_2[t]) \land \\
& (\forall t \in T_{\text{contra}} : \sigma_1[t] :> \sigma_2[t]) \\
\big) \lor \big(& \\
& \exists C_3 : d(C_1) \ \texttt{sub}\ C_3 \land C_3^{\sigma_1} <: C_2 \\
\big) &
\end{aligned}
$$

In the first alternative, we check whether the types correspond to the same class, i.e. whether they have the same *raw type* (type without type arguments). Additionally, all type arguments for invariant type parameters must be equal, type arguments for covariant type parameters must be subtypes, and type arguments for contravariant type parameters must be supertypes.

Using the second alternative, we gradually *lift* C_1 until it has the correct raw type by substituting all known type arguments into its declared supertype. We also use this lifting operation when inferring type arguments (Section 25.3).

Callable types. The final open entry from Table 14 is for callable types. Callables are contravariant in their parameters and covariant in their results[95] [169], but we must also consider the number of parameters and results, as well as parameter optionality.

Let F_1 and F_2 be callable types. F_i has m_i parameters and n_i results. $p_{i,j}$ denotes the j-th parameter of F_i and $r_{i,j}$ its j-th result. t is a function that looks up the type of a parameter or result, and o is a function that returns whether a parameter is optional. Then the subtyping rule "callable/callable" is

$$
\begin{aligned}
F_1 <: F_2 \Leftrightarrow\ & (m_1 \geq m_2) \land (n_1 \geq n_2) \land \\
& \big(\forall j \in [1 \,..\, m_2] : t(p_{1,j}) :> t(p_{2,j}) \land \big(o(p_{1,j}) \lor \neg o(p_{2,j})\big)\big) \land \\
& \big(\forall j \in [m_2 + 1 \,..\, m_1] : o(p_{1,j})\big) \land \\
& \big(\forall j \in [1 \,..\, n_2] : t(r_{1,j}) <: t(r_{2,j})\big)
\end{aligned}
$$

F_1 must have at least as many parameters and results as F_2 (line 1) and all extra parameters must be optional (line 3). All common results must be in a subtype relation (line 4). All common parameters must be in a supertype relation, and a parameter may not be required if it should be optional (line 2).

[95] This is also why the computation of in-/out-positions treats parameters and results as explained above.

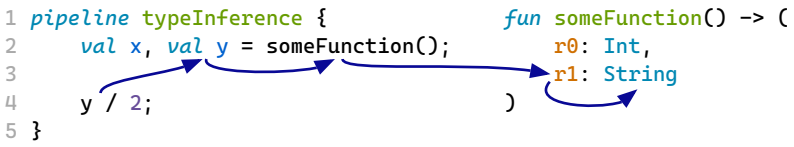


Figure 30: Inferring the type of a reference by traversing the AST until we find a manifest type.

25.3 Type Inference

Besides declaring types of parameters and results of segments, users of the pipeline language should not need to interact with the type system. To still offer type checking (Section 25.4), we must infer the types of all expressions as well as parameters and results of lambdas. This also necessitates inferring types for other language concepts, like placeholders, and looking up manifest types for attributes etc.

Basic type inference. For some language concepts, we immediately know their type:

- The type of parameters and results of named callables (any callable but lambdas) is their mandatory manifest type.
- The type of attributes is their mandatory manifest type.
- The type of literals is the literal type that contains them. For example, the type of `2` is `literal<2>`. We need to be this exact so we can check assignments to parameters with a literal type.
- The type of some operations is known immediately. The `===` operation (object identity), for instance, always has type `Boolean`.
- The type of other operations is known if the types of their operands are known. For example, the `*` operation (multiplied by) has type `Int`, if both their operands have type `Int`.

For anything else, we traverse the AST until we arrive at a base case. In line 4 of Figure 30, we want to check whether dividing `y` by `2` is legal. This requires that we infer the type of the reference `y`. To achieve that, we conceptually

1. go to the assignment that declares the referenced placeholder `y`,
2. determine the index of the placeholder (here 1; the placeholder `x` is at index 0),
3. go to the right-hand side of the assignment,
4. go to the called function and pick the result at the index from step 2, and
5. look up the manifest type of this result.

The downside of this approach is the need for state tracking during type inference, like the index of the placeholder, which must somehow reach step 4. We can solve this with the extra types we introduced for intermittent results of type inference (Section 25.1.2). Then we can infer the type of the call `someFunction()` to the named tuple type `(r0: Int, r1: String)`, regardless of where it occurs. When we infer the type of a placeholder, we simply look up the entry in the named tuple type at the corresponding index.

Judgments and type rules. The inference process can be formalized [169] by *judgments*, which have the form $\Gamma \vdash \mathfrak{J}$. Γ is a *static typing environment*, which is an ordered list of distinct declarations and their types. $\emptyset$ is an empty environment and $\emptyset, \texttt{r0} : \texttt{Int}$ is an environment that assigns the type `Int` to the declaration `r0`. $\mathfrak{J}$ is an *assertion* that is *entailed* by Γ. Γ needs to contain all declarations that are referenced in $\mathfrak{J}$. From this, we can build various judgments:

- $\Gamma \vdash \diamond$ denotes that Γ is well-formed, i.e. has the appropriate form.
- $\Gamma \vdash M : T$ is a *typing judgment* that denotes that the term M (e.g. some expression) has type T.

The validity of a judgment (*conclusion*) is defined by *type rules* based on the validity of other judgments (*premises*), which must all be valid. The validity of the premise judgments is again defined by type rules. At some point, one arrives at the *axioms*, which are judgments that are known to be valid. A suitable

$$\frac{}{\emptyset \vdash \diamond}\ \text{(Axiom } \emptyset\text{)} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{n} : \texttt{literal<n>}}\ \text{(Int lit) } (\texttt{n} \in \mathbb{Z}) \qquad \frac{\Gamma \vdash \texttt{M} : \texttt{Int} \quad \Gamma \vdash \texttt{N} : \texttt{Int}}{\Gamma \vdash \texttt{M * N} : \texttt{Int}}\ \text{(Int mult)}$$

Figure 31: Some type rules for concepts of the Safe-DS language. The judgments above the line are the *premises*, the judgment below the line is the *conclusion*.

axiom is $\emptyset \vdash \diamond$, which states that the empty environment is well-formed. A judgment is deemed valid, if it can be traced back to axioms, and invalid otherwise. This process is called *type derivation*.

Figure 31 shows an axiom and two other type rules. The name of the rule is written at the top, together with optional *annotations* like $\texttt{n} \in \mathbb{Z}$ that restrict its applicability. Next follow the premise judgments above the horizontal line. The conclusion judgment is written below the line. Via the rules "Int lit" and "Axiom $\emptyset$", we can then derive that $\emptyset \vdash 1 : \texttt{literal<1>}$ is a valid judgment.

As can be seen in Figure 31, connecting the "Int lit" and "Int mult" rules also requires the specification of judgments and type rules for subtype relations. The judgment $\Gamma \vdash T_1 <: T_2$ denotes that T_1 is a subtype of T_2 in the environment Γ. Type rules can then be derived from the rules in Section 25.2, by using the left operand of the equivalence relations as the conclusion and the right operand as the premises. If the right operand is a disjunction, one type rule must be defined per alternative. If the right operand is a conjunction, each argument becomes one premise judgment of a single rule.

The full formalization of the type inference requires the full specification of the syntax of the Safe-DS language, its scoping rules (where can a declaration be accessed), and type rules for all language concepts. As we still expect the language to evolve in the future, this is out of scope for this static document. Our implementation (Chapter 26) considers all these aspects. The language and type concepts we use, are already formalized in the existing literature (e.g. [169], [172]).

We instead focus on the two most interesting aspects of our type inference, inferring types of lambda parameters and type arguments.

Inferring types of lambda parameters. Unlike for named callable, the parameters of lambdas have no manifest types. But as lambdas must only be used as arguments of calls to named callables (Section 23.1), they are always *passed to a parameter with a manifest type*. This type must be a callable type. Thus, we can determine the types of lambda parameters by looking at the type of the corresponding parameter in the callable type.

Listing 23 demonstrates this, where line 1 contains a call to a higher-order function and line 3 the corresponding stub. We infer the type of the lambda parameter `row` by looking up the manifest type of the parameter `row` in the callable type. Matching happens purely by position, so the lambda parameter could also have a different name.

Inferring type arguments. To infer type arguments in constructor calls, we must consider subclassing, variance, and deeply nested parametrized types. For the sake of brevity, we will sketch how inference works if our type system just had class types and covariant type parameters and refer to [167] and [169] for a thorough discussion on the topic.

```
1    removeRows(       (row     ) -> row.getCell("age") == 19      )
2
3 fun removeRows(query: (row: Row) -> shouldRemoveRow: Cell<Boolean>)
```

Listing 23: Type of lambda parameters are inferred from the manifest type of the corresponding parameter in a callable type.

The basic operation we need is the computation of the *lowest common supertype* of a list of types, which is a subtype of all common supertypes of those types, regarding the subtype rules from Section 25.2. We compute it with the following pseudocode algorithm, where `types` is the list of types:

1. If `types` is empty, return the default type of the type parameter, or `Nothing` if it has none.
2. If `types` has length `1`, return the first entry.
3. Set `isNullable` to `true`, iff any of the `types` is nullable.
4. Set `candidates` to a list with the first entry of `types` and its direct/transitive supertypes[96].
5. Set the nullability of all candidates to `isNullable`.
6. For each `candidate` in `candidates`:
   a) If `candidate` is not a supertype of all `types` when ignoring type parameters, continue.
   b) If `candidate` has no type arguments, return it.
   c) Lift all `types` to the class of the `candidate`[97].
   d) For each type parameter of the `candidate` class, set the type argument in `candidate` to the lowest common supertype of the respective type arguments of the lifted types[98].
   e) Return `candidate`.

Computing type arguments for a constructor call is then done by calculating the lowest common supertype of the inferred types for arguments that are passed to a parameter of type `T`.

Let us go through an example, where our goal is to infer the types of the placeholders `a`, `b`, and `c`:

```
1 class Container<out T>
2 class Pair<out T>(a: T, b: T) sub Container<T>
3 class Scalar<out T>(a: T) sub Container<T>
4
5 segment mySegment(someInt: Int, someString: String) {
6     val a = Scalar(someInt);
7     val b = Pair(someString, someString);
8     val c = Pair(a, b);
9 }
```

For the placeholder `a`, we can derive the type argument already in step 2 because `T` only occurs once, and assign the type `Scalar<Int>` to the placeholder.

For the placeholder `b`, we check the candidates `String` and `Any`. `String` matches and has no type arguments, so we infer the type argument in step 6b), and assign the type `Pair<String>` to the placeholder.

For the placeholder `c`, we have the candidates `Scalar<Int>`, `Container<Int>`, and `Any`. As `Pair` is not a subclass of `Scalar`, this option is discarded. However, `Pair` is a subclass of `Container`. Since `Container<Int>` has type arguments, we lift all types to the `Container` class, which gives us the types `Container<Int>` and `Container<String>`.

We must then recursively compute the lowest common supertype of the type arguments `Int` and `String`. Here, we loop over the candidates `Int`, `Float`, `Number`, and `Any`, of which `Any` is the first supertype of `String` and our result. Plugging all results together, yields the type `Pair<Container<Any>>` for the placeholder c.

[96] A class stub declares its direct supertype after the keyword `sub`. The direct supertype must come first, then its supertype etc. This always includes `Any` at the end, the common supertype of all class types.

[97] This means walking up the inheritance chain up to the class of the `candidate`, while substituting type parameters.

[98] For contravariant type parameters, we must compute the *highest common subtype* instead. For invariant type parameters, the type arguments must be identical; otherwise we must try the next candidate.

25.4 Type Checking

The goal of type checking is to highlight mismatches between an expected type and an actual type at compile time, thus preventing a large class of runtime errors [169]. In particular, the actual type must be a subtype of the expected type, as explained in Section 25.2. The pipeline language uses type inference to determine actual types without user input (Section 25.3) and has type checking in the following places:

- **Arguments of calls**: The inferred type of an argument must be a subtype of the manifest type of the corresponding parameter.
- **Operands of operations**: Operations are essentially functions with special syntax. Hence, we also check whether the inferred types of their operands ("arguments") match their hard-coded allowed types ("parameters"). For instance, the `/` operation (divided by) may only be used on operands of type `Int`, `Float`, and `Cell`.
- **Results of lambdas**: The inferred type of a lambda result must be a subtype of the manifest type of the corresponding result of a callable type. For example, the lambda result in Listing 23 has type `Cell<Boolean>`, which is exactly the expected type. Checking parameters of lambdas is pointless, as their type is inferred from the corresponding manifest type, and, hence, always a subtype.
- **Default values of parameters**: The inferred type of the default value of a parameter must match the manifest type of the parameter. This is checked for segments in the Safe-DS language and any named callable in the stub language.

25.5 Constraints

Type checking already eliminates many runtime errors (Section 25.4), but callables may have preconditions that cannot be expressed by our type system. Hence, runtime errors can still occur. Where possible, we want to check these preconditions statically or at least without running the full code.

Preconditions of Python functions cannot generally be inferred automatically, since Python code is difficult to analyze statically due to dynamic imports and the lack of static type annotations, as discussed before[99]. Therefore, our stub language encompasses means to explicitly state preconditions via *constraints* related to numeric intervals, object state, and table schemas. More constraint kinds can be added in the future.

Boundaries. As shown in Table 8, many parameters of functions in scikit-learn are restricted to some interval. These constraints are defined by the domain and not specific to the implementation in scikit-learn. For instance, the number of trees in a random forest must be greater than 0.

In the stub language, the specification of a callable (function or class with constructor) may be followed by a `where` block that specifies a boundary constraint. A boundary constraint is a comparison relation between parameters of the described callable, or between a parameter and a constant.

To guarantee that the constraint can be statically evaluated, the parameters to which it refers must be marked by the `const` modifier, which requires that any argument passed to this parameter can be *partially evaluated* [54] to some constant value by our sophisticated partial evaluator. Based on such boundary specifications in stubs, adherence to bounds can be checked statically for calls made in the pipeline language.

The partial evaluator can handle expressions ranging from simple operations with literal operands, like `1 + 2` (which evaluates to `3`), up to complex calls of higher-order segments involving closures, as shown in Figure 32. Here, we determine the value of the placeholder `b` as `1` by traversing the AST until we arrive at a literal. The process is similar to type inference (Section 25.3), and could be formalized by the same judgment and rule notation, so we do not discuss it further.

[99] The automated stub generator (Section 24.4) extracts boundary constraints from docstrings, though.

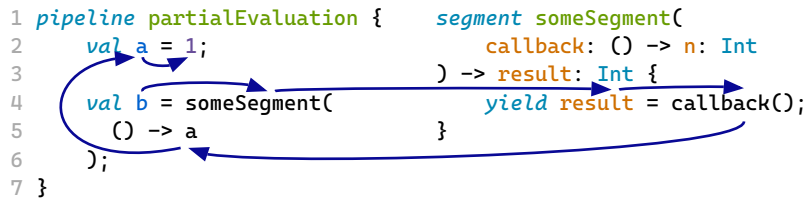


Figure 32: Inferring the value of a placeholder by traversing the AST until we find a literal.

```
1 class RandomForestClassifier(
2     const treeCount: Int
3 ) where {
4     treeCount > 0
5 }
```

Listing 24: A class with a *boundary constraint*. The parameter `treeCount` only accepts values larger than `0`. The `const` modifier ensures that only values are passed that can be evaluated to a constant.

Listing 24 has a boundary constraint to ensure that the `treeCount` of a `RandomForestClassifer` is larger than `0`. The call `RandomForestClassifer(treeCount = 0)` would, thus, cause a static error. Passing an argument that cannot be evaluated to a constant, like a call `f()` to a function stub `f`, also leads to a static error, because of the `const` modifier.

Boundary constraints are contained in a separate block rather than attached to individual parameters because a parameter may also be bounded by other parameters rather than a fixed value. In this case, it is cleaner to list boundaries separately than to attach it to one of the bounded parameters.

Object state. Whether calling certain methods on an object is legal, may depend on the state of the object. For example, calling the `predict` method on some `Classifier`, before calling its `fit` method is illegal, and effects a runtime error in the Safe-DS library.

Enumerations and bounded type parameters (Section 25.1) already facilitate tracking the state of an object with a finite state space statically, as illustrated by Listing 25. The enumeration `State` (line 1-4) defines all possible states a `Classifier` may be in.

The class `Classifier` has an optional type parameter to track its state (line 7). Only subtypes of the enumeration `State` are allowed as type arguments, and the default *type* (not value) of the type parameter specifies the initial object state, here the `Unfitted` variant. Thus, the inferred type of the placeholder `a` in line 15 is `Classifier<State.Unfitted>`.

The `fit` method meanwhile has a result of type `Classifier<State.Fitted>`[100], which is also the inferred type of the placeholder `b` in line 16. Note that the entire complexity of the state tracking is hidden in the stub code (lines 1-12) and does not appear in pipeline code (lines 14-17).

If `predict` was a global function, we could already restrict it to fitted classifiers using the type system with the following stub:

```
fun predict(classifier: Classifier<State.Fitted>, /* data omitted */)
```

The call `predict(a)` would then be detected as erroneous by the type checker, while `predict(b)` would be permitted.

However, `predict` is a *method* of `Classifier`, so we must restrict the availability of `predict` based on the type argument `S` of the receiver type. This cannot be expressed by the type system as the receiver of a method is implicit and, thus, has no manifest type. We could make the receiver explicit and let the

[100] This is another instance where returning new objects instead of mutating the call receiver helps.

```
enum State {
    Unfitted
    Fitted
}

class Classifier<
    S sub State = State.Unfitted
>() {
    fun fit(/* data omitted */) -> (
        fittedClassifier: Classifier<State.Fitted>
    )
}

pipeline myPipeline {
    val a = Classifier();
    val b = a.fit();
}
```

Listing 25: Tracking whether a classifier is fitted yet using an enumeration and a type parameter. The constructor of `Classifier` returns an unfitted classifier, while the `fit` method returns a fitted one.

type checker handle this case, but it would only be able to show a generic error like "Expected type `Classifier<State.Fitted>` but got type `Classifier<State.Unfitted>`", which does not convey the problem well and cannot offer a resolution.

Instead, we designed a new kind of constraint, the *state constraint* (Listing 26), and added the option to specify custom error messages for all kinds of constraints after the keyword `else`, which are displayed instead of auto-generated errors if the constraint is violated. The state constraint checks whether the type argument assigned to a type parameter of the containing class is a subtype of another type (Section 25.2).

Schema checks. When working with tables, we want to ensure that only columns that exist are accessed by users. The `collect_schema` method of Polars is already capable of computing the schema of a `DataFrame` or `LazyFrame` in the most efficient manner possible. Since our `Table` class is backed by these Polars classes (Section 17.2), we need no custom schema computer but can instead execute a Safe-DS pipeline with our runner (Chapter 31) to compute schemas.

The stub language only needs new kinds of constraints, e.g. to express that a certain column must be in a `Table`. An example is shown in Listing 27, where we check whether the receiving `Table` object (`this`) has a column with the given `name`. The value of `name` is again computed by the partial evaluator.

Proper *static* checking of table operations would require that users define schemas manually. Since we cannot check statically, whether the user-defined schema is correct, this would provide no safety gain. Moreover, manual specification of schemas is a time-intensive task and does not fit our goal to improve safety without burdening the user (Chapter 22).

```
    // Rest of the Classifier class ...

    fun predict(/* data omitted */) where {
        S sub State.Fitted else "Classifier is unfitted. Call 'fit' first."
    }
}
```

Listing 26: A *state constraint* that only allows calling `predict` on a fitted classifier. Otherwise, a custom error message is shown.

```
class Table {
    fun getColumn(const name: String) where {
        name in this
    }
}
```

Listing 27: A *schema constraint* that checks whether a table has some column.

26 Implementation (Language)

Acknowledgments. The stub generator has been implemented by Seyed-Arsam Islami as part of his ongoing Bachelor's thesis [173].

Progress and repositories. The current implementation of the Safe-DS language contains all features listed in prior chapters, except state and schema constraints (Section 25.5), which are future work. Custom error messages for constraints are supported. The implementation is open-source and available on GitHub[101].

Langium. The implementation is written in TypeScript and based on Langium [174], a tool that facilitates language development by deriving a set of services, like a parser and reference resolver, with reasonable default behavior, once a language's grammar and abstract model (classes that represent AST nodes for the language) are defined.

The default services can then be gradually overwritten by custom ones that exactly match the semantics of the language. For example, we changed the services responsible for reference resolution to implement the import system, including access to declarations in the same package or a `safeds.*` package without an explicit import. The type system and partial evaluator are implemented from scratch as custom services, since they are not available by default.

Program validity can be checked using a command-line tool[102]. This is beneficial for continuous integration, but not user-friendly. We will discuss an IDE for the Safe-DS language in Part V.

Language testing. To rigorously test all components of the language, like the type inferrer, type checker, and partial evaluator, we developed a system to define tests as normal Safe-DS files, so they can be created and edited quickly in our IDE (Part V). The system is based on special comments (`// $TEST$`) to denote assertions, and markers (»«) to denote locations in the code. Comments and markers are associated by position, i.e. the first comment belongs to the first marker etc.

For example, the following code tests the type inferrer and asserts that the type of a placeholder and the type of the expression on the right-hand side of the assignment are the same:

```
// $TEST$ equivalence_class assignedValue
// $TEST$ equivalence_class assignedValue
val »r« = »1«;
```

Automated stub generation. The stub generator extracts type hints from code using mypy [133]. Docstrings are handled by Griffe [175], which first transfers them into a common intermediate format[103], and then attempts to parse types contained therein. The stub generator is open-source and available on GitHub[104].

[101] https://github.com/Safe-DS/DSL
[102] https://www.npmjs.com/package/@safe-ds/cli
[103] As explained in Listing 1, Python docstrings can be written in various formats.
[104] https://github.com/Safe-DS/Stub-Generator

27 Related Work (Language)

Type checkers for Python. mypy [133], Pyright [176], and pyre [177] are type checkers for Python. They all require type hints [28]; otherwise, no checking is performed. All tools have a strict mode that raises errors if type hints are missing.

Schema checkers for Python. With pandera [178], users can manually describe the expected schema of `DataFrame` objects of various libraries, like pandas and Polars. Afterward, the schema can be used to validate whether some `DataFrame` object conforms to it.

A schema for a `DataFrame` contains names and types of columns. Moreover, arbitrary checks can be added to a column, which must pass for all values in the column. Schemas can be inferred, but inferred schemas should only be used as "rough drafts" according to the documentation of pandera, and require manual modification.

Full validation of a `DataFrame` only happens at runtime. An integration with mypy exists, but its usefulness is highly limited, as the schema of all `DataFrame` objects must be assigned manually via type hints and casts. No check whether a given `DataFrame` actually conforms to the assigned schema is performed statically.

Checking Python ML projects. mllint [179] is a static checker for Python ML *projects* (not individual pipelines). It runs various linters against the source code in the project, and checks whether tests exist, project dependencies are declared properly, version control is configured, and continuous integration is used.

Low-code configuration languages. Ludwig [180] lets users configure ML models using YAML files. A minimal configuration contains names and types of feature and target columns of the dataset that the model should be trained on. Based on this information, Ludwig derives a suitable model and training process. The default choices can be overwritten by additional configuration.

Ludwig focuses on simplicity, not safety. Still, Ludwig defines a schema for its configuration, which is checked early at runtime. If an illegal configuration value is detected, execution stops. Ludwig's runtime type system does not support type parameters, however, so values inside lists and dictionaries[105] are *not* checked early and can lead to type errors late into the execution. Moreover, users have no guidance from an IDE while they create the configuration, which easily leads to mistyped configuration names and values.

DS DSLs with other goals than safety. Various other DSLs exist for the DS domain that do not aim at improving safety:

- **Dataset description**: Giner-Miguelez et al. [181] developed a DSL for the manual description of ML datasets, which includes metadata (e.g. the authors), a composition part (e.g. various quality metrics), and a part for provenance and social concerns. With the SEMKIS-DSL [182], the requirements of neural networks regarding their inputs can be described. From the requirements, synthetic inputs can be generated. Lavoisier [183] derives code for data selection and formatting from dataset descriptions.
- **Big data**: Portugal et al. [184] compiled seven DSLs that handle challenges when working with big data. Pig Latin [185], for instance, provides an abstraction for the MapReduce programming model [186], and supports parallelized operations on vectors and matrices. For a comprehensive discussion of Pig Latin and the other six languages, we refer to [184]. FastML [187] is another, newer DSL that aims to hide the MapReduce model.

[105] These are the only containers of YAML, and thus often used.

- **Neural networks**: DeepDSL [188] supports automatic differentiation, and provides functions for NN layers, which can be composed to create complete NNs. TensorFlow Eager [189] represents TensorFlow computational graphs by imperative functions. StreamBrain [190] allows running Bayesian confidence propagation neural networks on various backends, like GPUs.
- **Other**: MD4DSPRR [191] aims at improving reproducibility by defining a separate logical model (intention) and operational model (implementation) of a DS pipeline. The tool verifies whether both models match. Arbiter [192] is a DSL for manual description of ethical concerns during model training, like the choice of features.

28 Future Work (Language)

Regarding the language and the stub generator, there is little work left. The implementation of the current concept is fairly complete and only lacks state and schema constraints (Section 25.5).

We also consider adding an *exhaustive switch expression*. It is not meant for control flow but for choosing the first value from a finite set, for which the associated condition is true. Say, we have a column containing numbers and want to create a new column that contains the sign of each number. With the exhaustive switch expression, it could look like this (design is not final):

```
column.transform((cell) -> switch {
    cell < 0  -> -1
    cell == 0 ->  0
    else      ->  1
})
```

29 Conclusion (Language)

In this thesis part, we presented the Safe-DS language, a textual DSL for the development of DS pipelines. It combines the expressiveness of Python DS libraries with safety via compile-time error detection, without extra effort from the user.

Python DS libraries are integrated into the languages by a *stub language* (Chapter 24), which accurately captures their API using type annotations (Section 25.1), and constraints (Section 25.5). Constraints can express preconditions of functions that go beyond the type system, like *boundary constraints* for numbers (to prevent out-of-bounds values), *state constraints* for objects (to ensure that methods are called in the correct order), or *schema constraints* for tabular data (to prevent illegal column accesses). Stubs can be created largely automatically from a library's code and documentation (Section 24.4).

DS pipelines are written in a simple *pipeline language* (Chapter 23) as a sequence of assignments and calls of functions defined in stubs. All types needed for type checking (Section 25.4) get *inferred* from the pipeline code (Section 25.3). This also applies to all information needed for constraint checking.

Our approach allows using the textual language as a serialization format for other kinds of user interfaces, like a graphical view. In fact, due to the simplicity of the language, we can even offer a *seamless, bidirectional transition* between the textual and graphical views. We will explore this possibility further in Part V, where we will also evaluate the DSL.

Overall, we hope that the Safe-DS language will prove valuable to help novices get started with data science and reduce development cost of data science applications in general.

IV Executing Data Science Pipelines

This part is a rewritten and extended version of a paper published previously as part of my PhD thesis:

- *An Alternative to Cells for Selective Execution of Data Science Pipelines* [193]

Abstract. Data scientists often use notebooks to develop data science (DS) pipelines. A pipeline is split into separate cells, which can be run individually, in any order. After a cell execution, the values of all variables contained in the cell are stored in the hidden state of the notebook, from where later cell executions can retrieve them.

However, notebooks for DS have multiple well-known flaws. We focus on the following ones in this thesis part:

1. Pipeline results depend on the order cells were executed in. Hence, sharing notebooks and input data is not sufficient for reproducibility.
2. After adding, editing, or deleting cells, the hidden state can become outdated. Users must then manually determine, which cells must be rerun. If they forget some or rerun them in the wrong order, the correctness of the result is compromised.
3. Even if program code does not change, external data might. However, notebooks do not detect this, again compromising correctness.
4. The efficiency of a pipeline depends on how well users understand the cell concept. If cells are too large, or users re-execute too many cells after an update (to prevent the second issue) much computing time is wasted on code that would not need to be re-executed.

To solve these issues, we suggest an execution concept that is *correct* (all required operations are run in the right order) but *minimal* (only the required operations are run). It is based on a fine-grained data-flow analysis that captures dependencies among individual operations and to external data. Significant additional speedups are achieved for pipeline re-execution by caching the results of functions. We also consider various strategies to discard cached values as memory runs out.

Explorative development of DS pipelines. The development of data science (DS) pipelines is typically an explorative process [3]. Developers often write or edit small pieces of code, rerun the code, and use the results to further change existing code or decide what to implement next. Due to the long runtime of DS pipelines, rerunning them entirely after each step is undesirable.

Notebooks, cells, hidden state. Notebooks have a solution for this: Developers write code in *cells* that can be executed independently, in any order. Notebooks maintain a *hidden state*, which is a mapping from variables to their current value. The hidden state is consistently updated when executing assignments. References to variables are then resolved to the current value in the hidden state (Figure 33).

Execution results, such as visualizations, are shown close to the code cells that produced them. Code cells can be interspersed with text cells, to offer explanations or document decisions, which follows the paradigm of literate programming [194].

Notebook variants. Various flavors of notebooks exist, with Jupyter Notebook [195] being most popular according to the 2022 Kaggle Survey on DS [102]. Jupyter Notebook requires language-specific

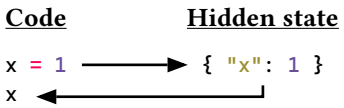


Figure 33: The *hidden state* of a notebook maintains a mapping from variables to their current value. Assignments update the internal state and references look up the current value.

kernels to execute code and to power features like code-completion, but is otherwise language-agnostic. Since Jupyter Notebook is a web application, it can run locally or be hosted as a service, like Google Colaboratory [196], Kaggle [197], or Amazon SageMaker [198]. Some integrated development environments (IDEs) like PyCharm [199] or Visual Studio Code [200] incorporate Jupyter Notebook. JupyterLab [201] is eventually meant to replace the default Jupyter Notebook GUI.

The issues we discuss in this thesis part are independent of a specific notebook variant, however, since they stem from the core concepts of notebooks: Cells and hidden state.

Running example. Figure 34 shows an example of the typical cell structure of a notebook for DS that contains Python code calling functions from the Safe-DS library (Part II). We use this program as a running example. For the sake of brevity, we omit text and result cells, which are irrelevant to our discussion. The three code cells

1. read a CSV file containing some data,
2. plot histograms, remove the columns `id` and `name`, and shuffle rows,
3. take the first ten rows as a sample and write it to another CSV file.

Based on Figure 34, we can illustrate the problems we want to discuss in this paper.

Problem 1: Outdated hidden state after code changes. Code changes partially invalidate the hidden state of the notebook [202]–[208]. Say, after executing the entire notebook from Figure 34, we add more data preparation steps after line 6 in cell 2. Now the value of `cleaned` in the hidden state is outdated, so cell 2 must be rerun. However, as `sample` was last computed from the outdated value of `cleaned`, its value in the hidden state is also outdated, so we must rerun cell 3, too.

Keeping track of all cells that must be rerun quickly gets complicated in large notebooks, leading to developers frequently rerunning *the entire notebook* to be safe [206], completely neglecting the potential runtime improvements from selective execution.

Problem 2: Outdated hidden state after data changes. Even if the code stays the same, the hidden state can get stale: If, for example, the file `titanic.csv` is changed on disk, the value of `titanic` (and all variables that depend on it) in the hidden state is no longer up-to-date. Hence, such an event

```
1 titanic = Table.from_csv_file("titanic.csv")                    (1)

2 plot = titanic.plot.histograms()
3 cleaned = (
4     titanic
5         .remove_columns(["id", "name"])                         (2)
6         .shuffle_rows()
7 )

8 sample = cleaned.slice_rows(length=10)
9 sample.to_csv_file("out.csv")                                   (3)
```

Figure 34: An example notebook with three code cells. It contains Python code that calls functions from the Safe-DS library (Part II).

necessitates partial re-execution of the notebook, but the developer must again manually track which cells need rerunning.

Problem 3: Wrong cell execution order. Users can execute cells in any order. However, not all orders are legal, let alone do all orders produce the same results [202]–[208]. For example, running cell 3 of Figure 34 first on a blank state is erroneous, as `cleaned` does not exist in the hidden state yet.

This might appear to be a purely hypothetical issue, as developers would clearly intend to run their cells from top to bottom. However, Pimentel et al. [205] analyzed 912,343 notebooks from GitHub with unambiguous execution order[106] and found that this issue drastically impacts reproducibility:

- 21% of the notebooks skipped execution of at least one cell. Running those notebooks top-to-bottom could alter results.
- 36% of the notebooks executed cells out-of-order. It is unclear whether this is the intended execution order or whether these notebooks are meant to be run top-to-bottom.
- 77% of the notebooks contained skips in the execution counter, so it is unclear which code was run in between and how it affected the hidden state.
- Overall, only 24% of the notebooks ran without exceptions when replicating the original execution order, and only 4% reproduced the result cells[107].

Problem 4: Unnecessary code execution. Since cells are the smallest unit of execution, even code that is unaffected by changes is re-executed. If we add data preparation steps after line 6 in Figure 34 and rerun cell 2, we always inadvertently recreate the histograms, even though the value of `titanic` is still valid in the hidden state (disregarding data changes). For larger datasets, this is an expensive operation.

The cell concept does not handle method chaining well, either. The entire chain (lines 3-7 in Figure 34) must be contained in a single cell, and thus always gets rerun completely, even if new calls are added to the end.

Overall, this drastically slows the feedback loop and leads to an *"expand then reduce" pattern* [203], where developers first write many small code cells (which can be executed independently) to iterate quickly and later combine them into bigger cells that reflect the logical structure of the code. This needs extra development time, though.

Goal: A new execution concept. In this thesis part, we want to design an execution concept that solves the problems outlined above. It should be *correct* (Problems 1-3), *reproducible* (Problem 3), and *minimal* (Problem 4).

Part outline. The remainder of this thesis part is structured as follows:

- Chapter 30 investigates state-of-the-art solutions for notebook issues and outlines the research gap.
- Chapter 31 refines our goals and covers the general ideas behind our own approach.
- Chapter 32 explains how we use the approach to run Safe-DS pipelines (Part III).
- Chapter 33 describes the implementation of the approach for the Safe-DS language.
- Chapter 34 evaluates our concept regarding runtime and memory requirements.

30 State of the Art: Notebook Execution

A natural solution to ensure code gets executed in the correct order (Problem 3) and gets re-executed after code changes (Problem 1) is to derive dependencies between cells. A cell `B` depends on a cell `A`, if `A` writes to a variable that `B` reads from. We also say that `B` is *downstream* of `A`, or `A` is *upstream* of `B`.

[106] Notebooks assign an increasing counter to each cell to indicate execution order. Ambiguity occurs if counters are repeated or if a cell is marked by a star, which only indicates that it is being run, but not when execution started.

[107] This is also partially due to unclear versions of the notebooks' dependencies, though.

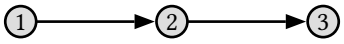


Figure 35: A *dependency graph* for the *cells* in Figure 34. An arrow from cell `A` to cell `B` indicates that `B` depends on `A`. We also say that `B` is *downstream* of `A`, or `A` is *upstream* of `B`.

Taken together, these relations build a *dependency graph*. Figure 35 shows the dependency graph for cells of Figure 34. Cells must then only be run after all cells they depend on, and must be rerun after a cell they depend on is rerun.

30.1 Existing Approaches and Tools

Various tools and approaches use this idea:

IPyflow. IPyflow (formerly NBSafety) [207] is a special kernel for Jupyter Notebook with two main modes: In the first, default mode (*reactive execution*), running a cell automatically reruns all its downstream cells, so their outputs stay up-to-date. In the second mode, cells that depend on stale hidden state are highlighted and the tool recommends which cells should be rerun to refresh the hidden state. Here, users must execute cells manually and may choose to ignore those suggestions.

IPyflow also has limited support for *memoization* [209], which means caching the results of calls for a set of arguments, and quickly returning results from the cache if known arguments are passed again. IPyflow uses memoization to avoid recomputing expensive cells. To enable memoization, users must add a special `%%memoize` marker to a cell[108].

marimo. marimo [210] is another tool for reactive execution of cells. After running a cell, all downstream cells are executed, too, or marked as stale, depending on user preference. Unlike IPyflow, it is a separate notebook implementation and not just a kernel for Jupyter Notebook. Because of this, marimo can also solve other issues of Jupyter Notebook, e.g. by offering dependency management, and storing programs as Python files[109] for better handling by version control tools.

Dataflow notebooks. Dataflow notebooks [211] ensures that all upstream cells have been run before a cell is executed. The tool assigns a persistent identifier to cells that stays the same even as cells are rerun or reordered. The tool remembers a mapping from the cell's identifier to its code upon last execution. It then compares a cell's current code to the stored one to determine whether it was changed and needs re-execution.

Nodebook. Nodebook [212] also reruns upstream cells as needed. This tool determines cell dependencies by the cells' inputs and outputs. Inputs are inferred statically by parsing the code: Any reference to a variable that is not declared in the cell itself, is considered an input. Outputs are discovered dynamically by comparing the hidden states of the notebook before and after a cell is executed. Any changed variable is considered an output.

ReSplit. ReSplit [213] analyzes dependencies between cells and definition-usage chains within the same cell and then suggests an alternative mapping of code to cells to ensure that tightly coupled code resides in the same cell.

Removing unused code. Head et al. [204] use cell dependencies to create a polished version of a notebook that only contains the code needed to produce the results that the user selected.

[108] Memoization is also an integral part of our execution concept. As we and the developers of IPyflow independently started implementation mid-November 2023 and arrived at diverging solutions, we will later provide our own motivation for memoization, without referring back to IPyflow.

[109] Jupyter Notebook stores programs as JSON files.

30.2 Research Gap

The existing approaches and tools provide solutions for Problem 1 (stale state after code changes) and Problem 3 (wrong cell order). However, none of them take changes to values outside the notebook into account (Problem 2). Moreover, they still use cells as the smallest unit of execution, so they fail to solve Problem 4, too. Users must still carefully manage their cell structure for optimal performance, e.g. by using the "expand then reduce" pattern, or code gets rerun unnecessarily, which slows the feedback loop.

31 General Approach for Partial Execution

Goals. As mentioned in the introduction, we want an execution concept that solves Problems 1-4. In particular, it must have the following properties:

- **Correctness**: We must run all code that is needed to calculate a result, and we must only run code once all its inputs are up-to-date. Rerunning a program should produce the same results as running it again from scratch. In particular, re-execution should not cause additional runtime errors.
- **Minimalism**: Only code that is needed to compute a result should be run.
- **Reproducibility**: Given a program, the external data it reads from, and an exact specification of a runtime environment[110], others should be able to obtain the same results as the original author. Correctness implies reproducibility, so we will mostly focus on the first two goals.

Outline. Our approach for partial execution of programs is based on a fine-grained dependency graph (Section 31.1), which is used to create a correct, and minimal execution plan (Section 31.2). To further accelerate program re-execution, we also cache the results of functions (Section 31.3).

31.1 Fine-Grained Dependency Graph

Operations as minimal unit of execution. To achieve minimalism, we must stop using cells as the minimal unit of execution. Instead, our execution concept works on the level of individual *operations*. For our discussion, we define an operation as a call or assignment, but the notion can be extended to other language concepts, like loops or conditional statements, too.

Basic dependency graph. Given a program, we first compute a fine-grained dependency graph between operations. An operation `B` depends on an operation `A` if running `A` affects the semantics of operation `B`. Figure 36 shows a first attempt at a dependency graph for Figure 34 that contains all variables (💾 symbol) and calls in the program. Assignments are shown by an arrow to a variable, and references by an arrow from a variable. Arrows between calls indicate a direct use of a call result without assigning it to a variable by chaining or nesting calls.

However, Figure 36 only captures calls and variables, which immediately raises several questions:

- How do we handle potential non-determinism (e.g. `shuffle_rows`)?
- How do we deal with potential object mutations (e.g. `remove_columns`)?

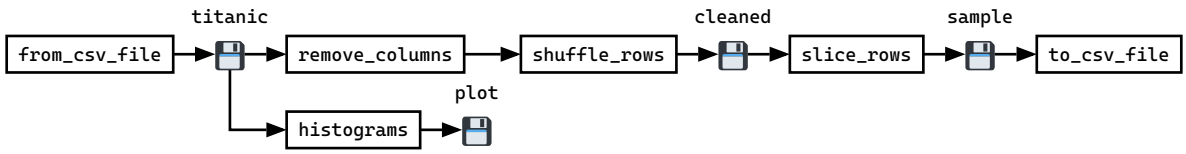


Figure 36: A basic dependency graph for the *operations* in Figure 34. Boxes stand for calls, and the 💾 symbol for variables. An arrow to the 💾 symbol indicates an assignment to a variable. An arrow from the 💾 symbol indicates a reference to a variable. Arrows between boxes show chaining or nesting of calls. An arrow from operation `A` to operation `B` indicates that `B` depends on `A`.

[110] This includes the operating system, dependencies, etc. A Docker image [214] would be suitable, for example.

- How do we control global state?
- What happens if an operation writes to a file that another operation later reads from?

Handling non-determinism. For the sake of reproducibility, all operations must be deterministic. The execution concept cannot help here, since non-determinism could have arbitrary sources, like race conditions, or access to random number generators. Hence, this must be solved on the library level. In scikit-learn [33], [34], for instance, many functions have a parameter `random_state`, which sets the seed of the random number generator and makes the function deterministic. All functions of the Safe-DS library, including `shuffle_rows` are deterministic.

Handling object mutations. Similarly, operations could mutate their inputs. In this case, rerunning a program only in part, might cause errors that do not appear when running it from scratch, violating our correctness goal.

This also affects notebooks: If the method `remove_columns` in Figure 34 would mutate its receiver `titanic`, running cells 1, 2, and then 2 again would raise an error because `titanic` no longer has the columns `id` and `name`, which should be removed.

Hence, to guarantee correctness for a program with mutating operations, we must either always execute it from scratch or make deep copies of all inputs to an operation before running it. Both options are slow, and making deep copies needs a prohibitive amount of memory. Thus, it is up to libraries to ensure they do not mutate objects, which was one of our core guidelines for the Safe-DS library because of this (Section 16.1.2). As the libraries know the semantics of an operation, they can use efficient techniques like copy-on-write [215].

Handling global state. Global state suffers from the same issue as object mutations and can cause subtle bugs or crashes when partially rerunning a program. Therefore, libraries must avoid it (which the Safe-DS library does).

Moreover, all variables are global state. If we reassigned the variable `titanic` in line 3 of Figure 34 instead of creating a new one, executing cells 1, 2, and 2 would again trigger an error. Such reassignments can be prevented on a language level, as done by the Safe-DS language.

Handling external data. Unlike the previous three issues, we *can* narrow down interaction with external data to a few sensible cases: A function might

- read from a file at a constant path,
- read from a file at a path given by one of its parameters (e.g. `from_csv_file`),
- write to a file at a constant path, or
- write to a file at a path given by one of its parameters (e.g. `to_csv_file`).

Hence, we can capture file dependencies between operations, i.e. where an operation reads from a file that another operation writes to. This leads to the enhanced dependency graph in Figure 37, where files are shown by the 📄 symbol, file reads by arrows from this symbol, and file writes by arrows to this symbol.

Summary: Assumptions about libraries and languages. To summarize, our execution concept only guarantees correctness, minimalism, and reproducibility under the following assumptions:

1. The library must only have deterministic functions.
2. The library must not mutate objects.
3. The library must not change global state.
4. The library can interact with files, but the files used by a function must either be specified by a parameter of this function or be at a constant path.
5. The language must prevent reassignments.

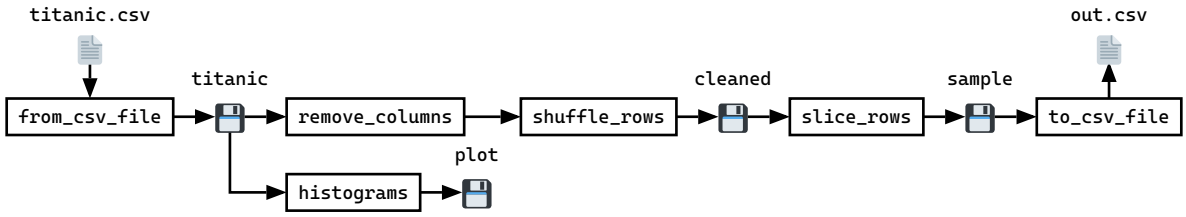


Figure 37: An enhanced dependency graph for operations in Figure 34 that also captures file interactions. The symbol stands for a file, an arrow from the symbol indicates a file read and an arrow to the symbol is a file write.

The state-of-the-art solutions (Chapter 30) must also make Assumptions 1-3 to fully work. They do not consider files at all, so Assumption 4 is not applicable. Dataflow notebooks allows reassignment, but to access previous values of a variable, users must append the identifier of the cell that contains the assignment to the name of the variable. This essentially boils down to using unique variable names in the first place, which we find more readable.

31.2 Initial Program Execution

Value inspection. In notebooks, users can run an individual code cell, which then shows its outputs in a generic result cell. But we want to run individual operations, not cells, so we need another approach for partial program execution and the display of outputs.

Our core idea for partial execution is to run a program *until some operation* in the dependency graph (Figure 37). In particular, we decided that users should only be able to run a program *until some assignment* because users are typically not interested in intermediate results of chained or nested calls[111] and selecting the exact target in a complex expression would be difficult for users. For example, we could run the program in Figure 34 until the assignments of `titanic`, `plot`, `cleaned`, or `sample`.

When running a program for the first time, executing a program in a correct, and minimal manner, is trivial. Say, we want to run the program in Figure 34 until the variable `cleaned` is assigned. We then start at the node for the variable in the dependency graph (Figure 37) and work our way backwards. Any operations we encounter must be executed, all others can be skipped. The operations that must be run are referred to as the *backward slice* [216]. We must execute upstream operations first, which leads to the execution plan in Figure 38.

The value of the computed variable is then displayed in a view that is designed for this type of data, which will be covered in Part V. For example, we will look at a specialized explorer for tabular data in Section 39.3.

File updates. Additionally, we need to consider running a program for its side effects, in particular to update all files that it writes to. Computing an execution plan works largely the same as for value inspection, except that we might have multiple starting points when computing the backward slice.

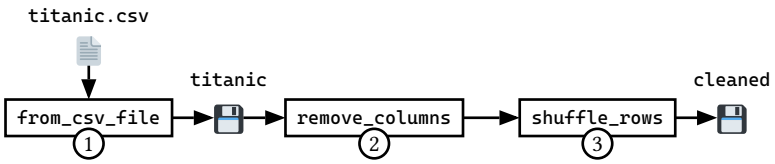


Figure 38: The plan for partial execution until the `cleaned` variable is easily derived from Figure 37. All upstream operations must be run, all others can be skipped. Upstream operations must be run first.

[111] In notebooks, these intermediate results cannot be inspected, either.

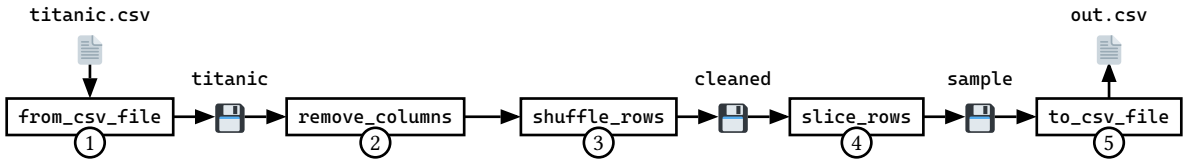


Figure 39: The plan for partial execution to update all files. Again, all upstream operations must be run, all others can be skipped. Upstream operations must be run first.

Hence, we compute the backward slice for all file writes and combine the results. Figure 39 shows the execution plan for the program from Figure 34.

31.3 Program Re-Execution

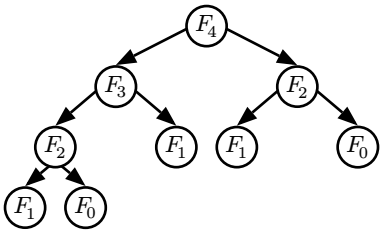
Partial re-execution of notebooks relies on their hidden state. However, as explained in the introduction, the hidden state can easily get stale as code is changed and users must then manually determine which cells to rerun.

Classical use of memoization. Our concept, thus, entirely gets rid of state that can get outdated. Instead, we rely on *memoization* [209]: Memoization is a technique used frequently in functional programming languages, like Haskell [217], to accelerate recursive functions [218]. A classical example is the computation of the nth Fibonacci number, which is defined by the following recurrence relation [219]:

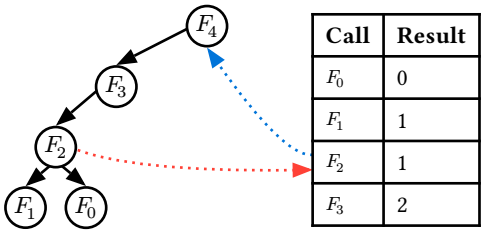
$$F_n := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

A naive recursive implementation is extremely inefficient because it recomputes the same results many times[112], as visualized in Figure 40 a). A memoized implementation, shown in Figure 40 b), instead stores all computed results in a table and later retrieves them from there when they are needed again, skipping the recomputation of the result.

Memoization between program runs. In the Fibonacci example, memoization is used to avoid expensive recomputations *during a single program run*. We use memoization to prevent expensive recomputations *between program runs*.



a) A naive implementation computes the same results many times.



b) A memoized implementation writes results to a table (red arrow) and reads them from there (blue arrow) once they are needed again.

Figure 40: Comparison of a naive and memoized computation of the fourth Fibonacci number.

[112] In fact, if $c(n)$ is the number of calls needed to compute the nth Fibonacci number, then $c(0) = 1$, $c(1) = 1$, and $c(n) = c(n-1) + c(n-2) + 1$ for $n > 1$. Hence, $\forall n \in \mathbb{N}_0 : c(n) \geq F_n$, i.e. the Fibonacci sequence is a lower bound for the number of calls needed to compute it this way.

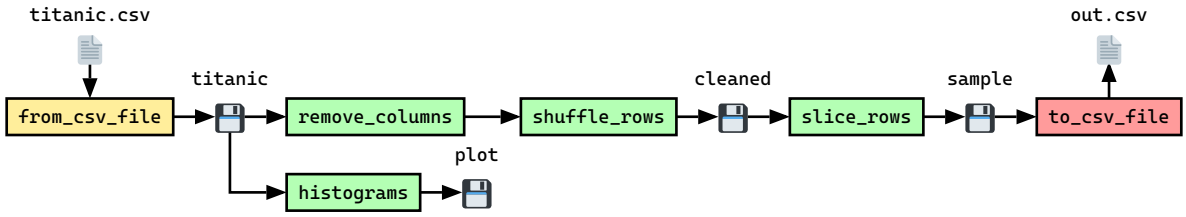


Figure 41: An augmented dependency graph with information about the purity of functions. Impure functions with side effects are red (`to_csv_file`), impure functions without side effects are yellow (`from_csv_file`), pure functions are green (the others).

As calls are executed, we store their results in a memoization table, which is maintained even after program execution is finished. Afterward, the developer can arbitrarily change their program. During subsequent re-execution of the program, we immediately retrieve results of calls with known arguments from the memoization table.

Requirements for memoization. However, memoization is only applicable to *pure functions*, which are functions that

- have no side effects, like writing to a file, and
- always return the same results for a given set of arguments.

Since the call is not executed again when the result is already contained in the memoization table, all side effects would be skipped, which incorrectly alters the semantics of the program. Likewise, if the function is, say, non-deterministic, looking up a previous result from the table might yield different program behavior than running the call again. As explained in Section 31.1, our execution concept only considers file reads and file writes as reasons for impurity.

Figure 41 show the purity of the functions that are called in our running example from Figure 34. The function `from_csv_file` is impure because it reads from a file, but only has an argument for the file *path*. Between program runs, the contents of the file could change, so the same call could produce a different result. The function `to_csv_file` is impure because it writes to a file as a side effect. All other functions are pure.

Implications of impurity. Not being able to memoize functions that write to files is fine because calls to these functions typically occur as leaves in the dependency graph. Hence, they are rarely part of the backward slice when running a program up to some variable (Section 31.2). However, functions that read from files commonly occur at the beginning of DS pipelines and are, thus, upstream of most operations in the dependency graph, which causes issues:

The function `from_csv_file` returns a new `Table` object each time it is called, even for identical CSV files. Thus, to know whether the result of the call to `remove_columns` already exists in the memoization table, we would need to compare *the entire data* of `titanic` with the data of all receivers of the memoized calls of the `remove_columns` function. This can be slower than just recomputing `remove_columns`.

Extended memoization table. To still be able to memoize calls to impure functions without side effects, we need to determine suitable *hidden arguments*, which are not part of the code, but which unique identify results together with the manifest arguments.

For functions that read from files but are otherwise pure, the *last modified timestamps* of the accessed files are suitable hidden arguments. If a function reads from a constant file, the timestamp alone is enough to uniquely identify the result. If a function reads from a file specified by an argument, the combination of the path and the timestamp is again sufficient.

Table 17: The extended memoization table after running the execution plan from Figure 38. For each call, it stores the called function, the arguments of the call, and its results. Impure functions that only *read* from files can be memoized by adding last modified timestamps of the accessed files. Identical colors for argument and result tables indicate the same object is referenced.

Function	Arguments		Hidden Arguments	Results
`from_csv_file`		`"titanic.csv"`	`2024-12-07T16:38:12`	
`remove_columns`		`["id", "name"]`		
`shuffle_rows`				

Table 17 shows the resulting extended memoization table after running the execution plan from Figure 38. For calls to pure functions, no hidden arguments are needed. For the call to the impure `from_csv_file` function, we include the timestamp of the read file.

Extended memoization in action. Because the result of `from_csv_file` is now also included in the memoization table, rerunning the execution plan from Figure 38 again is extremely fast, if `titanic.csv` was not changed:

1. Looking up the result of the `from_csv_file` call is successful and fast, as the arguments and hidden arguments are easy to compare.
2. Looking up the result of the `remove_columns` call is again successful *and fast* because we can now compare *object identities* of the `Tables` instead of the contained data.
3. The same applies to the call of `shuffle_rows`.

Meanwhile, this execution approach is correct and, in particular, always produces the same result as running a program from scratch (which also makes it reproducible): By definition, the memoization table does not contain results that could become stale. If the file `titanic.csv` is changed, its timestamp is changed, which causes `from_csv_file` to be rerun before a downstream operation is executed.

The execution approach is also minimal: If users run the program until some other variable, like `sample`, or change the code, previously computed values are still looked up quickly from the memoization table and only the missing results are computed (and then cached, where possible).

31.4 Freeing Memory

Since the memoization table is maintained between program runs, it needs increasing amounts of memory. Thus, we must also consider how memory can be freed again. For this, we additionally store various statistics for each entry:

- **Time to compute**: How long did it take to compute the results? This is set when the entry is created.
- **Memory requirement**: How much memory would be freed if this entry was removed? This is set when the entry is created.
- **Access timestamps**: When was the entry looked up? This is set when the entry is created and updated whenever it is looked up. We keep a list of all access times.
- **Times to compare**: How long did it take to compare arguments and hidden arguments? This is updated whenever the entry is looked up. We keep a list of all comparison times.

Based on these statistics, we define four basic strategies for the deletion of entries:

- **Least recently used (LRU)**: Remove the entry that has been unused the longest.

- **Most recently used (MRU)**: Remove the entry that was used last. This removes entries for downstream operations and keeps the entries for upstream operations.
- **Lowest time saved**: Remove the entry with the lowest difference between time to compute and the average time to compare.
- **Worst use of memory**: Remove the entry with the lowest ratio between time to compute and memory requirement.

We keep deleting entries from the memoization table using these strategies until the table fits the memory constraint. We will evaluate the performance of these strategies in Section 34.3. By default, the deletion strategy "worst use of memory" is used, but this is configurable.

31.5 Parallel Execution of Operations

The dependency graph also shows whether operations must be run sequentially or whether they may be executed in parallel. Sequential execution of operations `A` and `B` is needed, if there is a path from `A` to `B` (or vice versa) in the graph. Otherwise, the operations can be run in parallel.

For example, Figure 37 shows that we can run the calls to `histograms` and `remove_columns` (as well as all operations downstream of this call) inside the running example (Figure 34) in parallel.

We did not pursue this idea further, as we primarily want to use the execution concept to run Safe-DS program, which is also the topic of Chapter 32. The Safe-DS language integrates the Safe-DS library, which provides functions that are already parallelized internally. Starting more threads or processes would have little benefit and might even be harmful due to the additional overhead. But in other scenarios, running independent operations in parallel could further decrease execution times.

32 Executing Safe-DS Pipelines

We now explain how we apply the combination of a fine-grained dependency graph and extended memoization to run Safe-DS pipelines. Both elements require information about the purity of functions, in particular whether they read from or write to files. This information is manifest in the stub language (Section 32.1), and inferred in the pipeline language (Section 32.2). We also attempted to automatically infer the purity of Python functions (Section 32.5).

To run a given Safe-DS pipeline up to a certain point, the relevant backward slice is first transpiled[113] to Python (Section 32.3), and opportunities for memoization are marked in the generated code. A runner then executes the Python code and dynamically handles memoization (Section 32.4).

32.1 Manifest Purity in the Stub Language

The stub language only contains interfaces of functions without their implementation. Hence, inferring the purity of functions from stubs alone is impossible. As we will see in Section 32.5, purity inference from Python code is impractical, too. Therefore, purity information is manifest in the stub language.

For this, we implemented two annotations, `@Pure` and `@Impure`. One of these must be applied to each function stub. The `@Impure` annotation further expects a list of reasons for impurity, which are defined by variants of an enumeration that map to the file interactions from Section 31.1:

- `FileReadFromConstantPath`
- `FileReadFromParameterizedPath` (path given by a parameter)
- `FileWriteToConstantPath`
- `FileWriteToParameterizedPath` (path given by a parameter)

Listing 28 shows the resulting stubs for `fromCsvFile`, which is impure because it reads from a file given by its `path` parameter, and `shuffleRows`, which is pure.

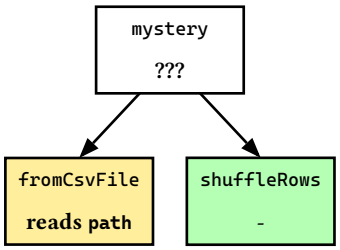
[113] Transpiling means to translate code from one high-level language to another.

```
class Table {
    @Impure([ImpurityReason.FileReadFromParameterizedPath("path")])
    static fun fromCsvFile(path: String) -> table: Table

    @Pure
    fun shuffleRows() -> newTable: Table
}
```

Listing 28: Safe-DS stubs with purity annotations. The method `fromCsvFile` is impure because it reads from a file (given by its `path` parameter), but `shuffleRows` is pure.

```
pipeline callGraph {
    mystery();
}

segment mystery() -> table: Table {
    val titanic = Table.fromCsvFile("titanic.csv");
    yield table = titanic.shuffleRows();
}
```

a) A simple Safe-DS pipeline that calls a segment.



b) The computed call graph.

Listing 29: Inferring the purity of the `mystery` call by building a call graph with purity information. Green nodes are pure functions, yellow nodes read from a file.

32.2 Inferred Purity in the Pipeline Language

In the pipeline language, we do not want to burden developers with adding purity information. Hence, the purity of each call is inferred, just like types (Section 25.3), and information for constraint checking (Section 25.5). Inference is done in two steps:

1. Compute a call graph. Declarations in the pipeline language that do not contain calls and declarations in the stub language are the leaves of the call graph.
2. Propagate impurity reasons through the call graph. A call is pure if there are no impurity reasons; otherwise it is impure.

A simple example. We explain this process using the Safe-DS pipeline in Listing 29 a). Our goal is to infer the purity information for the call to the `mystery` segment. The segment only contains two calls, both to stubs, so we compute the small call graph in Listing 29 b).

Since we only call stubs, we can immediately add their manifest purity information to their nodes in the call graph: `fromCsvFile` reads from a file that is specified by its `path` parameter, and `shuffleRows` is pure. Note that we can treat stubs as black boxes, since we assume that all their reasons for impurity are manifest.

To infer the purity of the `mystery` call, we aggregate the impurity reasons from its downstream nodes. Here, it is only the file read caused by `fromCsvFile`. But, as we now know the argument that is assigned to the `path` parameter of `fromCsvFile`, we can instead propagate a file read from the constant path `"titanic.csv"` as the impurity reason. This replacement is done if our partial evaluator (Section 25.5) can evaluate the argument to a constant string. In the end, we correctly determine that `mystery` is impure due to the file read.

Recursive propagation in complex call graphs. For more complex call graphs, the propagation is done recursively in a depth-first manner. We track visited nodes to avoid endless recursion if the call graph has loops. As the pipeline language has no conditional execution, this always indicates endless

recursion in the analyzed program, so we add an extra impurity reason "endless recursion" to the call that started the loop. The call is then flagged as erroneous, preventing program execution.

32.3 Code Generator

The Safe-DS language enables safe use of Python libraries. Thus, to run a Safe-DS pipeline, we must convert it to Python. This is done by a *code generator* that gets

- the file that contains the pipeline,
- all files it depends on (e.g. because of imports), and
- a list of target statements up to which the pipeline should be run.

The target statements either assign placeholders or write to files, as explained in Section 31.2. The code generator then

1. computes the backward slice for the target statements,
2. traverses the AST nodes in the backward slide in a depth-first manner, and
3. creates Python code for each node.

Composition of Python code. The Python code for an inner AST node is typically a simple composition of the Python code for their child nodes. For example, the Safe-DS code `1 + 2` is translated to the Python code `(1) + (2)`, by first generating code for the operands (`1` and `2`) and then substituting the result into the template for the full operation (`($leftOperand) + ($rightOperand)`). The extra parentheses ensure the correct precedence for more complex operands.

Handling imports. Imports are a notable exception: Required Python imports are collected during code generation and then added at the top of the generated file as a final step. This prevents duplicate imports and allows us to sort and group imports. Translating Safe-DS imports directly to Python imports would not work, as imports to files from the same package or a `safeds.*` package can be omitted in Safe-DS, but not in Python. The `@PythonModule` annotation may alter which Python module gets imported (Section 24.2).

Handling block lambdas. Block lambdas are another exception, as they do not exist in Python. Instead, they are translated to a *nested* function definition (to maintain the closure semantics) and a pointer to that function definition (Listing 30). The name prefix `__gen_` is reserved in the Safe-DS language and must not be used in names of declarations. This prevents name collisions in the generated Python code.

Mapping references and calls to Python. References to declarations are translated to Python as explained in Section 24.2: By default, the Safe-DS name is used in the generated Python code. For declarations with a `@PythonName` annotation, the name specified as an argument there is used instead. For calls to functions with a `@PythonMacro` annotation, the code is created by substituting the template expressions in its template (Listing 31).

Source maps. Even though the Safe-DS language catches many categories of errors statically, runtime errors are not fully avoidable. These runtime errors would point to lines in the generated Python

```
1 aggregate((a, b) {
2     yield sum = a + b;
3 });
```

a) Safe-DS code

```
1 def __gen_lambda_0(a, b):
2     return a + b
3
4 aggregate(__gen_lambda_0)
```

b) Generated Python code

Listing 30: Block lambdas do not exist in Python, so we must instead translate them to a nested function definition and a function pointer.

```
"Hello, world!".length()
```

a) Safe-DS pipeline

```
class String {
  @PythonMacro("len($this)")
  fun length() -> length: Int
}
```

b) Safe-DS stub

```
len(("Hello, world!"))
```

b) Generated Python code

Listing 31: A call to a Safe-DS function with the `@PythonMacro` annotation is translated to Python by substituting template expressions in its template.

```
Table.fromCsvFile(
    "titanic.csv"
);
```

a) Safe-DS code

```
safeds_runner.memoized_static_call(
    "safeds.data.tabular.containers.Table.from_csv_file",
    Table.from_csv_file,
    [safeds_runner.absolute_path("titanic.csv")],
    {"separator": ","},
    [safeds_runner.file_mtime("titanic.csv")]
)
```

b) Generated Python code

Listing 32: The code generator tells the runner (Section 32.4) which calls can be memoized. For statically dispatched functions, the code generator includes the unique qualified name of the function (line 2), a pointer to the function to call (line 3), the positional arguments (line 4), the named arguments (line 5), and the hidden arguments (line 6).

code, which is unhelpful. Therefore, the code generator also creates source maps [220], which map the generated Python code back to the Safe-DS code it originated from. This allows associating runtime errors with the Safe-DS code that caused them.

Memoization. Lastly, the code generator decides which functions to memoize based on the inferred purity of calls (Section 32.2). Memoization is then controlled at runtime by the runner (Section 32.4). Communication between the code generator and the runner happens via an API provided by the runner that is called in the generated code.

Depending on whether the function is dynamically dispatched [52] (instance methods) or statically dispatched (global functions and static methods), different code is created. Listing 32 shows the generated Python code for a call to the static method `fromCsvFile`.

The runner API offers helper methods to resolve a relative path to an absolute path based on the current working directory (`absolute_path`), and to get the last-modified timestamp of a file (`file_mtime`). Using these building blocks, the code generator then tells the runner how to memoize the call to `fromCsvFile` using the `memoized_static_call` function and passing as arguments

1. a unique identifier for the called function (its qualified name),
2. a pointer to a function to call,
3. a list of the positional arguments of the call,
4. a map of the named arguments of the call, which includes the default values of optional parameters that are not overwritten, and
5. a list of hidden arguments.

In Section 32.4, we explain how the runner executes this generated code.

32.4 Runner

The *runner* is responsible for executing the code that was created by the code generator (Section 32.3). Since the code generator outputs normal Python code, the runner can simply run it with any Python

interpreter. The runner must only provide the API used by the code generator, like the `absolute_path`, `file_mtime`, and `memoized_static_call` functions. The first two are trivial, but we will look at `memoized_static_call` in more detail:

As a reminder, memoization avoids repeated computations by storing the results of functions in a memoization table. The function `memoized_static_call` implements this principle using the following pseudocode algorithm:

1. Create a memoization key from the qualified name, positional arguments, named arguments, and hidden arguments. It must uniquely identify a call.
2. If the memoization table has an entry for the memoization key, return the associated result.
3. Build a call from the function pointer, positional arguments, and named arguments and execute it.
4. Store the computed result in the memoization table.
5. Return the result.

For Listing 32, the call created in step 3 would be `Table.from_csv_file("/path/to/titanic.csv", separator=", ")`. Hidden arguments are only used for the memoization key, but not in the call.

The runner also has a function `memoized_dynamic_call`, which is largely identical, but supports dynamic dispatch. Along the way, `memoized_static_call` and `memoized_dynamic_call` compute the statistics required to remove entries later (Section 31.4), too.

32.5 Purity Inference for Python

Naturally, it would be helpful if the stub generator (Section 24.4) would automatically add purity annotations by analyzing the Python implementation of a function, building a call graph, and propagating purity information. But, as noted previously, Python is not well suited for static analysis, due to dynamic imports and lack of static type information, and DS libraries are not well suited for dynamic analysis, due to long runtimes.

Purity inference must be precise. If the analysis cannot accurately determine the purity of a function `f`, it must mark the function as impure for indeterminate reasons (which then propagates through the call graph and taints other functions). Such a function can never be memoized, as it might have side effects.

This is not too bad if the library fits our assumptions (Section 31.1). Then, we at least know that the function at worst reads from or writes to files, so most other optimizations still work properly: We can memoize functions other than `f` as usual and only need to include `f` in the backward slice of operations that appear *after* a call to `f` in the pipeline and that read from any file.

But to guarantee correctness in all cases, we must assume that `f` can have arbitrary side effects, like changing global state or mutating objects, unless proved otherwise. Then, none of our optimizations work, and we must always run the program from scratch without memoization.

In short, if we consider pure to be the positive class, the purity inferrer must have *no* false positives for correctness (100% precision), and as few false negatives as possible for efficiency (high recall).

Precise purity inference for Python is not useful. Lukas Radermacher implemented a static purity inferrer for Python as part of his Bachelor's thesis [221], but it is not suitable for practical use. On the Safe-DS library, it achieved the required 100% precision, but only 65.8% recall. There are several contributing factors to the low recall:

- For efficiency, many libraries implement functions in languages other than Python (e.g. Rust for Polars) and only provide Python bindings. Hence, the Python code contains no implementation that could be analyzed.

- Functions that are built into Python are also often not implemented in Python, either (e.g. `len`), making analysis impossible. For the most commonly used functions, Radermacher manually specified their purity in a lookup table.
- Due to dynamic imports and the lack of static types, we cannot accurately identify call targets[114]. Radermacher filtered functions by their name and arity, but this still left many candidates. To ensure correctness, the impurity reasons of *all* candidates had to be combined.

Overall, this is not a limitation of Radermacher's tool, but of static analysis of Python code in general. Thus, purity information must be added manually to stubs. This is typically little effort for a library that already meets our assumptions (Section 31.1). The Safe-DS library, for example, only has 12 impure functions, so we let the stub generator mark functions as pure by default and then overwrite the purity as needed.

33 Implementation (Execution Concept)

Acknowledgments. The code generator and runner were implemented by Winston Oberländer during his Bachelor's thesis [222].

Progress and repositories. The execution concept for Safe-DS is fully implemented as described in Chapter 32. The purity inferrer, slicer (to compute the backward slice), and code generator are integrated into the repository for the Safe-DS language[115]. The runner is in a separate public repository[116].

Outline. In the remainder of this chapter, we explain the architecture of the runner (Figure 42) and challenges that occurred during its implementation. The other components of the execution system were straightforward to implement and will not be discussed further.

IDE/Runner Communication via WebSockets. We want users to trigger program execution from the integrated development environment (IDE) for the Safe-DS language. As we will see in Chapter 40, this IDE is implemented in TypeScript. But since the runner must offer a Python API for the code generator, it makes sense to fully implement the runner in Python. Thus, we need a means for the TypeScript and Python components to communicate. We use WebSockets [223] for this, which allow efficient full-duplex communication between a client and server.

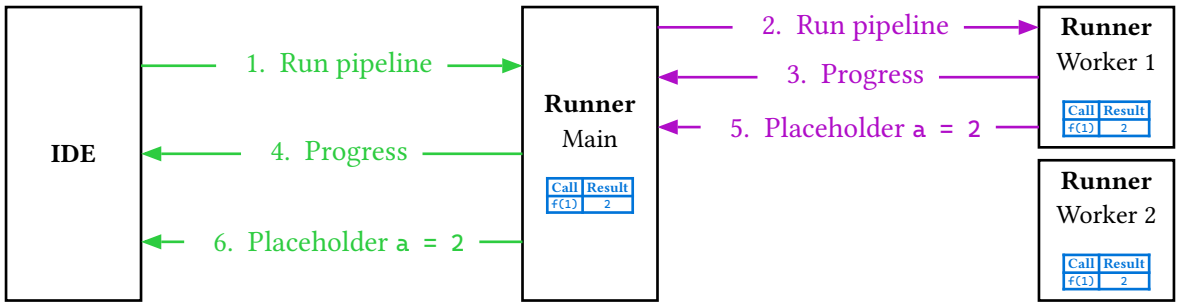


Figure 42: The IDE and runner communicate via WebSockets (green). The runner has a main process, which only handles WebSockets communication and delegates the actual program execution to worker processes. Communication between the main and worker processes happens via a message queue (purple). All processes can access the memoization table (blue) via shared memory.

[114] Even if we have static types, we cannot *uniquely* identify the target of method calls because of dynamic dispatch. But static types would reduce the number of candidates to consider.

[115] https://github.com/Safe-DS/DSL

[116] https://github.com/Safe-DS/Runner

The client is part of the IDE, while the server is part of the runner. The client sends the Python code that was generated for a Safe-DS pipeline to the server, which then runs the program and sends progress messages, errors and warnings during program execution, and the values of requested placeholders (Section 31.2) back to the client.

Worker processes for execution. To support parallel execution of several Safe-DS pipelines at once, the main process of the runner only handles WebSockets communication, but delegates program execution to worker processes. The main process and workers communicate via a message queue. The maximum number of workers to spawn is configurable. If all workers are busy, the program is queued, until a worker is available again.

Shared memory for the memoization table. All worker processes must be able to read from and write to the memoization table. Oberländer tried various implementations for the coordination between workers. The use of shared memory [224] proved to be the most efficient option, since processes only need their own copies of the *pointers* to the objects in the memoization table. Communication between processes is minimized.

Explicit object identity. The use of shared memory causes another problem, though: As mentioned in Section 31.3, we want to look up entries in the memoization table by comparing object identities, i.e. pointers (`a is b` in Python). Comparing objects structurally (`a == b`) could be slower than recomputing the object. But if different workers have their own copies of the pointers, comparing object identities would always fail.

Hence, we make the object identity explicit by assigning our own Universally Unique Identifier (UUID) [225] to an object before adding it to the memoization table. Then we can instead compare these explicit object identities, which are shared between processes.

Looking up entries in the memoization table. Result lookup from the memoization table happens in two stages:

1. We compare explicit object identities. This quickly finds a matching entry, should it exist, and fails fast if no object has the same identity.
2. We compare objects structurally[117]. This typically fails fast for objects that are not equal structurally (e.g. `Table` objects with different schemas or number of rows, or `Image` objects with different sizes). However, proving that two objects are structurally equal can be expensive.

The second step is only worthwhile, if the comparison is faster than recomputing the result, or if we can then quickly lookup results of downstream operations by comparing explicit object identities. Based on our evaluation results (Section 34.3), we are considering removal of the second step in the future. This would lead to more cache misses, but might improve runtimes overall.

34 Evaluation (Execution Concept)

We evaluated the execution concept according to three research questions, which will be answered in the subsequent sections:

- RQ1: How is execution time affected by using a fine-grained dependency graph for operations? (Section 34.1)
- RQ2: How is execution time affected by memoization? (Section 34.2)
- RQ3: How is execution time affected by the various strategies for freeing memory (Section 34.3)

[117] Two objects with different identities could still be equal structurally.

34.1 Effect of Fine-Grained Dependency Graphs

Our approach calculates the backward slice using a fine-grained dependency graph for *individual operations* (Figure 37), while all state-of-the-art tools compute a dependency graph between *cells* (Figure 35). All competing tools always run a cell completely, even if only a few statements in a cell are needed.

Therefore, the runtime of a program in a cell-based tool heavily depends on how the program is separated into cells. If the entire program is contained in a single cell, the execution time is maximized. If each operation is contained in its own cell, execution time is minimized.

To answer RQ1, we executed the running example (Figure 34) using marimo[118] in three different cell configurations[119]:

1. One cell for the entire program.
2. Three cells, as shown in Figure 34.
3. Six cells, with one cell per call.

In each case, we first ran the program up to the `to_csv_file` call. Afterward, we added `"ticket"` to the list of columns to remove (line 5) and ran the program again[120]. For each run, we aggregated the execution times that were reported for each executed cell in the notebook.

We then repeated the same experiment using the execution system for Safe-DS (Chapter 32) on an equivalent Safe-DS pipeline[121]. Since we do not use cells, there were no further variants of this experiment. Memoization was disabled for this test. We measured the entire round-trip time from the code generation until the display of the result, which also subsumes all steps from Figure 42.

All experiments were repeated ten times on a laptop with an Intel i7-8650U CPU, 16 GB of RAM, and Windows 11 Pro. The aggregated results are reported in Table 18.

In the 1-cell and 3-cell configurations, marimo always runs the expensive call `plot.histograms`. During the first run, this also triggers initialization of PyTorch, which takes several seconds. Safe-DS never runs this call, as it is not contained in the backward slice of the `toCsvFile` call, making it faster by an order of magnitude.

In the 6-cell configuration, marimo is slightly faster than Safe-DS during the first run, as the runner first starts its workers (which are then kept alive). This is a fixed cost, that is independent of the run pipeline. It is also a few milliseconds faster when rerunning a program, since we have some overhead for code generation, and WebSockets communication. Again, this is largely a fixed cost, though.

Table 18: Aggregated runtimes of the example notebook (Figure 34) in milliseconds using our execution approach for Safe-DS, and when running it with marimo in various cell configurations.

Statistic	First Run				Rerun after Change			
	marimo 1 Cell	marimo 3 Cells	marimo 6 Cells	Safe-DS	marimo 1 Cell	marimo 3 Cells	marimo 6 Cells	Safe-DS
Mean	5295	4967	300	440	423	457	21	32
Median	5260	4952	295	439	422	429	22	30
Std Dev	304	137	15	27	67	68	6	9

[118] This is currently the most popular tool we found according to GitHub stars. Other tools behave identically.

[119] The `"titanic.csv"` file is available online: https://datasets.safeds.com/en/stable/datasets/titanic/

[120] A change is needed to trigger re-execution in marimo.

[121] Note that in both cases, the Safe-DS library was used, so if there was no slicing, the same Python code would be run.

Overall, we conclude that the fine-grained dependency graph of our execution concepts lets us run Safe-DS pipelines as fast as state-of-the-art tools for running Python notebooks (plus a small constant overhead), without relying on an unrealistic cell structure. For suboptimal cell structures, which are bound to happen in larger pipelines, our approach is considerably faster. We offer efficiency as default.

34.2 Effect of Memoization

The effect of memoization (RQ2) was evaluated during Oberländer's Bachelor's thesis [222]. He implemented seven popular Python DS pipelines from Kaggle, which varied greatly in complexity and the size of the input data, in the Safe-DS language. He then fully ran each Safe-DS pipeline four times without changing the pipeline between runs. This experiment was done once without memoization and once with memoization on a laptop with an Intel i7-7700HQ CPU, 16 GB of RAM, and Windows 10.

Table 19 contains the measured runtimes in seconds for the first run (without and with memoization) and the mean of the three reruns (again without and with memoization). It also compares the size in Megabytes of the input files on disk to the size of the memoization table in memory.

The runtimes for the first run indicate that memoization causes practically no overhead. Sometimes program execution is even faster with memoization than without, but this is likely caused by random fluctuations in hardware performance and certainly no advantage of memoization.

When rerunning a program without changing it, memoization drastically reduces runtimes for all but the most simple pipelines ("Titanic Tutorial"). In the case of the "Credit Fraud" pipeline, for example, rerunning the pipeline with memoization saved more than 21 minutes and was faster by a factor of 212. The full memoization table still easily fits into memory for input files of more than 100MB.

Naturally, the reported results only apply under the unrealistic circumstance that a program is not changed at all between runs. Each change lessens the effect of memoization. If we ran an entirely unrelated program, memoization would not improve runtimes, but this is an equally unrealistic scenario.

Table 19: Effect of memoization on the runtime (in seconds) of Kaggle pipelines during the first run and repeated runs. The table also lists the size of the input files on disk and the size of the memoization table in memory in Megabyte. Large improvements due to memoization are highlighted.

Pipeline	Time First Run (s)		Time Rerun (s)		Size (MB)	
	No Memo	With Memo	No Memo	With Memo	Input Files	Memo Table
Titanic Tutorial[122]	4.48	6.24	4.47	4.10	0.09	1.26
Titanic High Accuracy[123]	21.46	23.57	21.30	4.65	0.09	24.25
Titanic Ensemble Stacking[124]	23.92	22.62	23.71	8.01	0.09	16.06
Regularized Linear Models[125]	56.30	56.01	55.77	5.46	0.91	63.53
New Job[126]	331.89	312.25	335.90	4.66	2.17	150.83
Beginner Classification[127]	1222.15	1214.76	1223.45	99.34	127.90	1158.59
Credit Fraud[128]	1323.14	1257.21	1299.82	6.13	150.83	590.11

[122] https://www.kaggle.com/code/alexisbcook/titanic-tutorial
[123] https://www.kaggle.com/code/ldfreeman3/a-data-science-framework-to-achieve-99-accuracy
[124] https://www.kaggle.com/code/arthurtok/introduction-to-ensembling-stacking-in-python
[125] https://www.kaggle.com/code/apapiu/regularized-linear-models
[126] https://www.kaggle.com/code/khotijahs1/predict-who-will-move-to-a-new-job
[127] https://www.kaggle.com/code/archaeocharlie/a-beginner-s-approach-to-classification
[128] https://www.kaggle.com/code/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets

Table 20: Effect of strategies for freeing memory by deleting entries from the memoization table on pipeline runtimes (in seconds). Full memoization (last column) is best in all cases, but would not match the memory constraint. The next best option that fits into memory is marked.

Pipeline	LRU	MRU	Lowest time saved	Worst use of memory	No Memo	With Memo (No Deletion)
Titanic Tutorial	6.87	6.52	6.65	6.72	4.47	4.10
Titanic High Accuracy	23.71	8.29	12.40	10.90	21.30	4.65
Titanic Ensemble Stacking	21.67	17.37	12.13	12.25	23.71	8.01
Regularized Linear Models	104.84	83.69	132.25	71.34	55.77	5.46
New Job	320.95	211.54	22.43	40.73	335.90	4.66
Beginner Classification	1258.69	1049.66	1061.82	700.63	1223.45	99.34
Credit Fraud	712.90	1003.52	658.55	903.84	1299.82	6.13
Mean	349.95	340.08	272.32	249.49	423.49	18.91

34.3 Effect of Strategies for Freeing Memory

Oberländer [222] also investigated the effect of the different strategies for freeing memory (Section 31.4) on pipeline runtimes (RQ3). For this test, Oberländer used the same Safe-DS pipelines from Section 34.2 and again ran them four times on his laptop without making changes between runs.

For each pipeline, he restricted the maximum available memory to 75% of the size of the full memoization table (Table 19), forcing deletion of entries in all cases. The pipeline "Titanic Ensemble Stacking", for instance, was only given 12MB of memory for the memoization table.

Note that this is a very strict test of the worst case, where we cannot even memoize all the results of a single pipeline run. Ideally, memory should suffice for the results of a single pipeline run, and we would only delete entries that are no longer needed anyway because the pipeline code was changed.

The mean runtimes of the three reruns are reported in Table 20. For reference, Table 20 also repeats the original runtimes without and with memoization from Table 19. Using no memoization would also match the memory constraints, while using full memoization would not, by design of the experiment.

Table 20 shows that even in this extreme case, memoization with any strategy for freeing memory outperforms running the pipelines without memoization on average. However, it is far slower than using full memoization, emphasizing that having enough memory to store a single pipeline run is crucial for efficient explorative development of DS pipelines[129].

Out of the analyzed strategies, "lowest time saved" and "worst use of memory" were the most promising options, with "worst use of memory" performing best on average. Because of this, we made "worst use of memory" the default strategy.

Nevertheless, in two cases, using no memoization at all was the best choice. Here, entries for intermediate operations are dropped from the memoization table, so they get recomputed in subsequent runs. Then, objects do not have the same explicit identity, but are equal structurally. This leads to lookup times, which are longer than recomputing the results from scratch.

More research is needed to find the ideal strategy for deleting values and to determine whether falling back to comparing objects structurally is worth it in general, or whether we should only compare explicit object identities and recompute operations if we find no matching entry this way.

[129] This also applies to notebooks, where values assigned to variables are stored in the hidden state in memory.

35 Related Work (Execution Concept)

Our dependency graphs for operations are directed and acyclic. Such *directed acyclic graphs* (*DAGs*) are used by other tools as the basis for execution, too.

Build tools. Gradle [226] is a build tool for Java, Kotlin and other languages. A build workflow is defined by *tasks*, such as running the Java compiler. Each task explicitly states

- which other tasks it depends on,
- which files or directories it uses as input, and
- which files or directories it produces as output.

Based on the task dependencies, Gradle builds a DAG to ensure that downstream tasks are only run after upstream tasks. Moreover, task execution is skipped if inputs are unchanged (*incremental compilation*), similar to how we handle external files in our execution concept. Other build tools offer similar features.

Apache Airflow. Apache Airflow [227] allows the definition, execution, and scheduling of arbitrary workflows, like backing up some hard drive. A workflow (also called *pipeline*) is composed of tasks that depend on each other.

Workflows and tasks are written in Python using decorated function definitions. Task dependencies can either be defined explicitly (`A >> B` means that task `B` depends on task `A`) or implicitly via function calls. In the latter case, users would call `B` with the result of `A` as argument and Apache Airflow would then infer an edge from `A` to `B` in the DAG to indicate the dependency[130]. This implicit interface is similar to how we derive dependencies between operations.

Tensor libraries. Tensor libraries, like PyTorch [115] and TensorFlow [117], define calculations using directed, acyclic *computational graphs*. The libraries offer primitive operations, like defining a constant tensor, or multiplying matrices. These can be composed to larger computations by passing the result of an operation `A` as an argument to another operation `B`. This defines an edge from `A` to `B` in the computational graph. The libraries use the computational graph to compute backward slices and to run independent operations in parallel. We use the same principles, but apply them to entire DS pipelines.

36 Future Work (Execution Concept)

Better strategies for freeing memory. The benefit of memoization drops when entries are deleted from the memoization table. Thus, we must carefully decide, which entries to remove. The strategies, we suggested in Section 31.4 are a start, but the evaluation showed that none of them are perfect yet (Section 34.3).

Hence, designing better deletion strategies is a possible future research direction. For example, we could consider the following strategies:

- **Lowest number of downstream entries**: Remove the entry that the fewest other entries depend on. An entry `A` depends on an entry `B`, if `A` uses the result of `B` as an argument. Removing `B` would mean that we could no longer access `A` by quickly checking explicit object identities.
- **Lowest cumulative time saved**: We define the *cumulative time saved* as the sum of the differences between compute time and average compare time for an entry and all downstream entries. We would remove the entry with the lowest cumulative time saved. This might be a more exact variant of the previous strategy “lowest number of downstream entries”.
- **Combining time saved & memory use**: The strategy “worst use of memory” uses compute time as the numerator. Instead, it could use time saved or cumulative time saved. This might combine the advantages of the two best performing strategies we tested (Section 34.3).

[130] The implicit definition only works for a small subset of Python, akin to the Safe-DS pipeline language.

Potentially eliminate lookup by structural comparison. As outlined in Chapter 33, we first look up entries by comparing object identities, and (if this fails) by comparing objects structurally.

However, for some pipelines it seems better to only compare object identities and recompute values if this fails (Section 34.3). Despite increasing the number of cache misses, this can improve runtimes if memory is tightly restricted.

The decision whether to keep or remove structural lookup needs a comprehensive benchmark, consisting of many Safe-DS pipelines of different complexity that operate on datasets of various sizes. To create this benchmark, we can continue translating Kaggle notebooks to Safe-DS to ensure that the contained examples are realistic.

The benchmark should also contain realistic change scenarios, so we do not only repeatedly run the same program. Then, the benchmark must be run for different memory constraints, e.g. a fixed amount of memory or some percentage of the maximum memory required for a single run.

Based on the results of the benchmark runs, we can then either disable structural lookup completely or derive rules when it should be active.

Cloud. The benefit of memoization heavily depends on the available memory[131] (Section 34.3). If the local system has insufficient memory, running the computation in the cloud could help. The architecture of the runner (Chapter 33) supports moving its main process and workers to the cloud in theory.

A blocking issue is the assumption that input files to a program already reside on the system that executes the runner. If the input files are on the local system, and the runner is in the cloud, this assumption fails. This is easily solvable, though, e.g. by adding messages to the WebSockets protocol between IDE and runner, so the runner can request a file, which is then sent by the IDE[132]. The code generator could replace the generated `absolute_file` calls (Section 32.4) with calls to another function that triggers this exchange.

Moreover, the runner can execute arbitrary Python code, so adherence to best practices for information security is critical when moving to the cloud. In particular, we need to provide access control and ensure the *CIA triad* [228], i.e.

- confidentiality of exchanged messages and the memoization table,
- integrity of all exchanged messages and the memoization table, and
- availability of the service.

37 Conclusion (Execution Concept)

In this thesis part, we presented an alternative to cells for the explorative development of DS pipelines. We derive a fine-grained dependency graph that focuses on individual operations rather than cells (Section 31.1). It also contains reads from and writes to files.

Users then define up to which point they want to run their pipeline. From this, we derive an execution plan that is *correct* (contains all operations and runs them in a valid order) and *minimal* (does not contain unnecessary operations) by determining a backward slice (Section 31.2).

While running a pipeline, we store the results of calls to pure functions and functions that are only impure because they read from files in a memoization table. When a pipeline is rerun, potentially after making changes, we look up previously computed results to avoid expensive recomputation (Section 31.3). We also suggest various strategies for the deletion of entries from the memoization table, so it fits into restricted memory (Section 31.4).

[131] Naturally, this applies to the maximum size of datasets that can be processed, too.

[132] For large data, other solutions might be more suitable, including the ability to fetch data directly from the web.

The approach is used to efficiently run Safe-DS pipelines, without any further effort from the user: The stub language declares the purity of functions (Section 32.1), so it can be inferred in the pipeline language (Section 32.2). Safe-DS code is then translated to Python (Section 32.3) and executed by a runner (Section 32.4).

Safe-DS pipelines can be run almost as quickly as state-of-the-art notebook tools can run equivalent degraded Python notebooks that only contain one call per cell (Section 34.1). We only have a small constant overhead for generation of Python code and communication with the runner. The addition of memoization leads to a further drastic reduction in execution times when rerunning programs up to a factor of 212 (Section 34.2), given sufficient memory (Section 34.3).

Overall, we expect that our approach will significantly accelerate DS pipeline development, by (1) avoiding bugs resulting from access to stale notebook state, and (2) avoiding unnecessary code execution. As code is always executed on a well-defined state and in a fixed order, our approach is also a major step toward reproducibility of results.

V An Integrated Development Environment for Data Science

This part is a rewritten and extended version of papers published previously as part of my PhD thesis:

- *Safe-DS: A Domain Specific Language to Make Data Science Safe* [143]
- *An Alternative to Cells for Selective Execution of Data Science Pipelines* [193]

Abstract. An *integrated development environment* (*IDE*) bundles features that facilitate program development. It can support the editing process with code-completion, display of errors in the code, automated refactorings, and more. Additionally, IDEs can aid program understanding. e.g. by syntax highlighting, tooltips, or visualizations for class hierarchies and call graphs. Frequently, IDEs also help with compiling, running, debugging, and versioning a program.

In this thesis part, we present an IDE for the Safe-DS language that offers most of these features. Moreover, the IDE complements the textual Safe-DS pipeline language with an equivalent graphical representation that further eases development of Safe-DS pipelines for DS novices by preventing syntax errors completely. Seamless synchronization of textual and graphic views lets developers choose the one best suited for their skills and current task, without getting stuck in a view.

Finally, we introduce a table explorer, which allows users to easily understand their tabular data without writing any code. Besides the data, the table explorer shows statistics for individual columns, alerts users of bad data quality, can create plots, and provides basic table operations, like sorting and filtering.

A world without IDEs (and AI assistants). Grace works as an intern for a startup in the financial sector. She has recently been tasked with implementing the business-critical login system in the backend. Unfortunately, she knows neither the language (Kotlin) nor the library (Ktor) she is expected to use. Nonetheless, she immediately launches Notepad, her favorite plaintext editor, and starts coding.

She quickly realizes that she could really use a second screen, as she keeps jumping between the editor, and the online documentations for Kotlin and Ktor to look up the syntax of the language and search for suitable API elements. Grace also decides to spend the next few nights on learning touch typing, as she gets frustrated by her slow typing pace.

Still, after persevering for two weeks (including weekends), she is done with the login system. After another few web searches and some experimentation, she manages to set up the Kotlin compiler and a runtime environment with the latest version of Ktor. Grace proudly enters the command to compile her code into a terminal — only to be greeted by hundreds of error messages.

Confused, she asks her colleague Ada for help, and explains her development process. Ada quickly spots the error: Grace has correctly installed the latest version of Ktor, which is also used by existing components of the project, but referred to the documentation for an outdated version. The APIs of the two versions differ drastically. Subserviently, Grace cancels her plans for the upcoming weekends, too.

Static checking can be too late. We argued in Part III that *dynamic checking* does not provide the desired safety during development, leading to long feedback loops before bugs are detected, particularly for long-running programs like data science (DS) pipelines. The short story above illustrates that

static checking can still be too late to prevent pointless work. There, the Kotlin compiler was only invoked after the entire faulty program was written already. Thus, what we really need for safety is *continuous static checking*, i.e. showing problems in the code immediately as it is being written.

Code-completion is a powerful aid. Static checking can only flag erroneous code that is already written. Grace likely would have never written the faulty code in the first place, though, if she had configured the project dependencies at the start and had access to code-completion while programming. Code-completion would only suggest API elements that are part of the installed version of Ktor.

Additionally, code-completion reduces the number of switches between the editor and documentation, as it can be used to find the correct API elements. Code-completion has the added benefit, that context can be considered, like the receiver of a member access, leading to far fewer suggestions than the static documentation contains. Finally, code-completion saves keystrokes, especially if the names of API elements are verbose.

From plaintext editor to IDE. Adding features like continuous static checking or code-completion to an editor gradually turns it into an *integrated development environment* (*IDE*)[133]. An IDE should make programming in a certain language as easy as possible. Ideally, it is the *only* tool that developers need for programming in that language.

To achieve this, an IDE may combine language-agnostic functionality, e.g. for text manipulation (multiple cursors) and version control, with language-specific features like syntax highlighting, automated refactorings [11], or the aforementioned continuous static checking and code-completion.

Goal: An IDE for the Safe-DS language. Since the Safe-DS language (Part III) is a new language developed by us, no IDE with language-specific features exists yet, which makes the implementation of Safe-DS pipelines impractical. Hence, this thesis part is dedicated to the design of a comprehensive and user-friendly IDE for Safe-DS.

It should facilitate development of Safe-DS pipelines, particularly by focussing on safety: Errors during development should be prevented or at least detected early. Additionally, the IDE should provide means to execute Safe-DS pipelines using the system presented in Chapter 32. This also implies that the IDE must provide views for value inspection, e.g. for tables.

Part outline. The remaining content of this thesis part is structured as follows:

- Chapter 38 investigates a selection of state-of-the-art tools that are used for the development of DS pipelines.
- Chapter 39 presents our refined goals and approach for the IDE.
- Chapter 40 discusses the current implementation of the IDE.
- Chapter 41 evaluates the IDE regarding usability and development speed.

38 State of the Art: Development Tools for DS

DS is a fast-moving field, which makes it impossible to describe even a significant fraction of the existing tools for the development of DS pipelines. We can exclude libraries and languages, as we already looked into those in Chapter 14 and Chapter 21 respectively. But we must further restrict the remaining tools to the three categories that are most relevant to the upcoming discussion, and only list prominent examples for each category. To our knowledge, these omissions have no effect on the research gap (Section 38.4):

- Section 38.1 covers generic Python IDEs.
- Section 38.2 considers graphical editors for DS pipelines and explains the benefits and disadvantages of this visual approach.

[133] There is no universally agreed upon minimum list of features an IDE must have.

- Section 38.3 describes how the tools from Section 38.1 and Section 38.2 present results to users (e.g. the data in a table). This section also discusses dedicated tools for data exploration.

38.1 Generic Python IDEs

Visual Studio Code (VS Code) [141] and PyCharm [62] are the most popular IDEs among the respondents to the 2022 Kaggle DS survey [102], with 41.8% and 25.4% regular usage respectively[134].

Language-agnostic features. Both IDEs offer comprehensive language-agnostics features, like a file explorer, multiple cursors to edit text in several locations at once, global search and replace (with support for regular expressions), an integrated terminal, and a GUI for version control systems like Git.

Language-specific features. Additionally, both tools provide features to ease development in a specific language, like Python, via official or third-party extensions. Simple extensions add context-free syntax highlighting for a language, or contain a collection of snippets, which are templates for language constructs (e.g. a class definition) that show up in code-completion. Snippets help developers learn the syntax of a language.

Complex extensions may add continuous static checking, code-completion, automated refactorings, GUI elements to execute or debug a program, and more. PyCharm offers a dedicated API for language support, but now also understands the Language Server Protocol (LSP) [229] that was introduced by VS Code. The LSP is a protocol for the communication between a language server and a language client.

The *language client* is a generic editor or IDE that knows nothing about a specific language. The *language server* provides tooling for a specific language, such as continuous static checking, but knows nothing about specific IDEs. This decoupling enables

- connecting multiple language servers for different languages to one language client, and
- using the same language server in different language clients, which greatly reduces the effort to support a language in multiple IDEs.

Notebooks. Finally, PyCharm and VS Code can handle Python notebooks. When editing a notebook, both IDEs provide the same features as for normal Python programs, which makes development more comfortable compared to using the web interface of Jupyter Notebook directly.

38.2 Graphical No-Code Tools

Graphical tools promise fast development of DS pipelines without writing any code. Instead, users define their pipeline by placing operation nodes on a canvas. The data-flow is defined by edges between the operations, which connect their outputs and inputs.

We already looked at Altair AI Studio in Section 21.1, but many other tools follow the same visual programming paradigm, including SageMaker Canvas by Amazon [230], Visual Blocks for ML (formerly Rapsai) by Google [231], Dataiku [232], KNIME [233], and Orange [234].

User interaction is fairly similar in all tools, so we do not further differentiate between them. To add operations, there are generally two options:

1. Users select operations in a global toolbox that contains all operations and drop them onto the canvas. The toolbox groups operations by the task they solve and offers a search. Afterward, users must manually connect output ports (results) and input ports (parameters) of operations with edges.
2. Users click on an output port of an operation, which opens a panel that contains only the operations that make sense in this context. After selecting an operation from this filtered list, output and input ports are connected automatically.

[134] Multiple selection was possible, so some respondents might use both.

Primitive inputs like numbers are not shown in the main graph, but can be inspected and edited by clicking on an operation, which either opens a sidebar or a floating window.

All tools provide large sets of predefined operations. Users can also define their own nodes, which run arbitrary Python code (or code written in another textual language). However, the tools neither provide IDE features for writing this code, nor can they visualize it[135].

Advantages. Compared to coding in a textual language, the graphical tools let users make fewer mistakes: Syntax errors and typos can only happen when editing the primitive inputs. Furthermore, users without programming experience might be more comfortable and familiar overall with a graphical tool.

Disadvantages. We already discussed the disadvantages of *purely* graphical tools in Section 21.1, so here is only a summary: Graphical tools suffer from

- a slower editing process compared to writing code (depending on the user's experience),
- hidden information (the graph quickly runs off the screen and primitive inputs are always hidden),
- lack of support for standard development tooling (e.g. version control with Git),
- tool lock-in, and
- vendor lock-in for closed-source, proprietary tools.

38.3 Value Inspection

DS pipelines are usually developed in an explorative manner (Part IV). An unfinished pipeline is frequently run, and users decide how to modify and extend the pipeline based on the results of each run. Because of this, user-friendly inspection of the results is important.

Basic notebooks. If a notebook code cell ends with an expression, the value of this expression is shown in a generic result cell after execution. The notebook environment defines renderers that nicely display values to users, e.g. for HTML markup. Classes define how their instances are supposed to be presented by implementing certain methods: For example, `_repr_html_` returns an HTML representation [235]. A class may define multiple representations. The final display format is chosen based on the available renderers and the representations the class offers.

Datalore. Datalore [236] offers a notebook interface with a dedicated renderer for tabular data. In separate tabs, users can

1. view, sort, and filter the data in the table,
2. create basic scatter plots, line plots, area plots, bar charts, and correlation heatmaps,
3. check statistics for each column.

The statistics tab always contains some generic information for each column, like the number of missing values. Otherwise, the displayed statistics depend on the type and distribution of data in the column, as can be seen in Figure 43. For categorical columns with many distinct values (`name`), no additional information is shown. If there are only a few categories (`sex`), the value distribution is displayed as a histogram. For numerical columns (`age`), Datalore presents several statistics and a combined box plot and violin plot.

Suspected problems with data quality are emphasized with warning markers (yellow background) or error markers (red background) depending on their severity. However, these problems cannot be resolved in the result view and instead require writing code.

[135] To visualize arbitrary Python code, tools would need to design a graphical equivalent for *all* language concepts of Python, including list comprehension, generators, and asynchronous functions, which would lead to a bloated, incomprehensible graphical language.

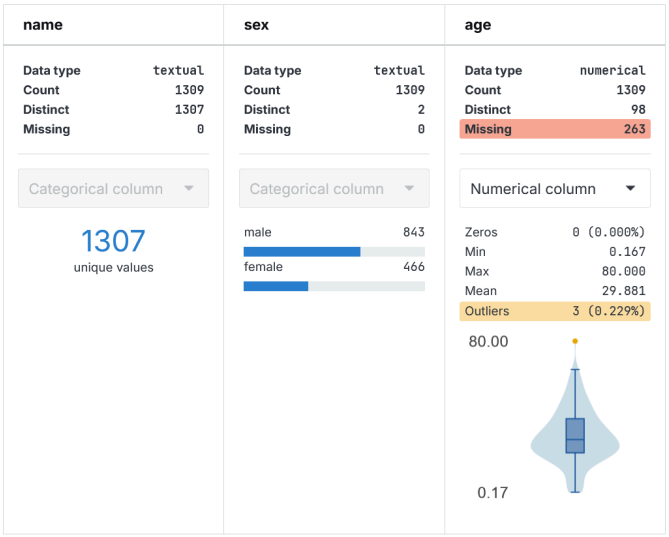


Figure 43: Statistics shown by Datalore for various types and distributions of data. Potential problems with data quality get highlighted. Some statistics for the `age` column were hidden for brevity.

Data Wrangler and SandDance. Data Wrangler [237] is an extension for VS Code that provides a dedicated explorer for pandas `DataFrames`. It can be triggered from result cells of notebooks or by right-clicking suitable files (e.g. CSV) in the file explorer of VS Code.

Data Wrangler has two modes: viewing and editing. The viewing mode combines the data of the table with statistics about each column. The data can be sorted and filtered. The editing mode additionally shows operations for data cleaning, e.g. to fill missing values (Figure 44). Data Wrangler keeps a history of all applied operations and can add a Python code cell that accomplishes the same effect with pandas to the notebook.

Data Wrangler has no operations to create custom plots for a `DataFrame` without writing code. This is instead handled by another extension for VS Code, SandDance [238], which can be opened by right-clicking on a suitable file (e.g. CSV) in the file explorer. SandDance cannot be triggered directly from a notebook. Instead, users must first write a computed `DataFrame` to a file, before they can open it with SandDance.

Graphical tools. Tools for visual programming of DS pipelines have no cells, so they need dedicated GUI elements for value inspection. Orange and KNIME display results of operations in the same dialog, where primitive arguments of operations can be modified. Figure 45 shows this principle for histograms, but there are also views for other plots, tables, images, and more.

It is easy to regenerate plots after changing primitive arguments in Orange and KNIME. However, changing plot types (e.g. from histogram to box plot) is slightly cumbersome, as users must close the old edit dialog, add a new node for the plot to the graph, and open a new edit dialog. Dataiku solves this by having a dedicated, integrated plot view, where users can choose (and later change) a plot type, and enter arguments for the selected plot.

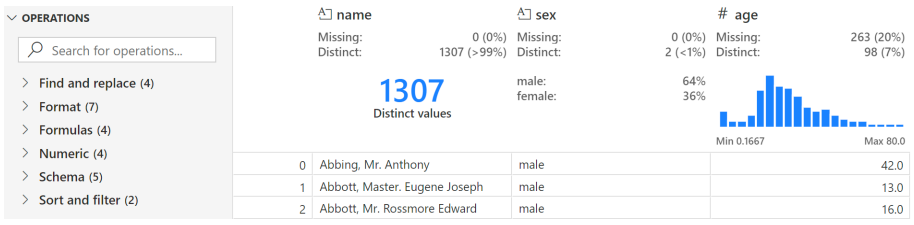
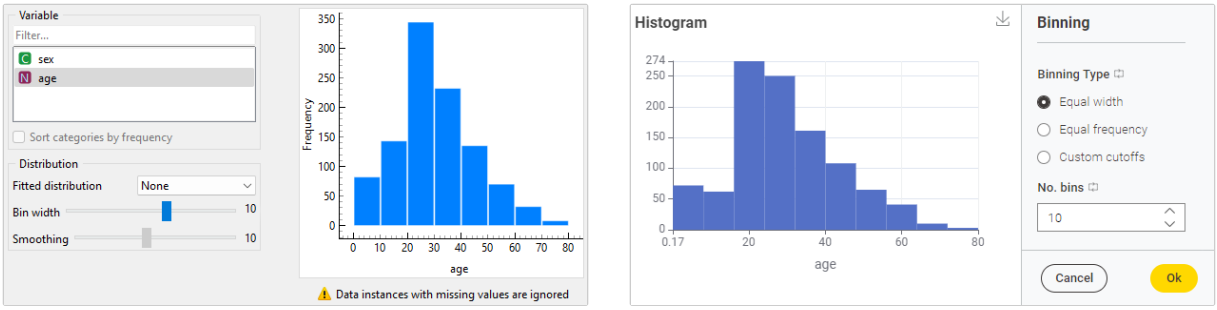


Figure 44: Data Wrangler has a combined view for statistics and data and can edit data.



a) Orange b) KNIME

Figure 45: Graphical tools for DS pipelines have dedicated GUIs for value inspection. Orange and KNIME integrate results into the dialog for changing primitive arguments of an operation.

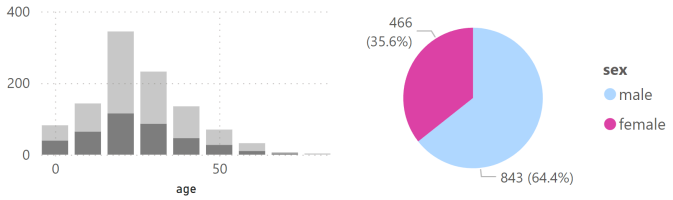


Figure 46: An interactive dashboard created by Power BI. Clicking on plot elements (like the `"female"` slice in the pie chart) selects the corresponding portion of the data and updates other plots.

Business intelligence tools. Such visualization builders are also a core feature of business intelligence tools, like Tableau [239] and Power BI [240]. The created visualization can be combined to interactive dashboards (Figure 46). Clicking on plot elements selects the corresponding part of the data. For example, clicking on the `"female"` slice of the pie chart in Figure 46 selects all rows of the connected table where the value of the `sex` column is `"female"`. This selection affects all other plots for the same table, too. The histogram in Figure 46, for instance, overlays the age distribution for women (dark gray) on top of the age distribution for all people in the table (light gray).

38.4 Research Gap

As we saw in previous sections, there are sophisticated tools for textual development and for graphical development of DS pipelines. Both textual and graphical tools have powerful features for value inspection, too.

However, there is no tool that allows seamless switching between a textual and graphical view. Such an approach would combine the benefits of a textual view (fast development for programmers, compact presentation, use of standard development tools like Git, etc.) with the benefits of a graphical view (ease of use for non-programmers, and added safety), as users can always pick the view that best fits the current situation and change views if needed.

Moreover, to make development of pipelines with the Safe-DS language practical and profit from the safety added by the language (Part III), an IDE is required. Since the language is new, no IDE exists yet. Running Safe-DS pipelines with the execution concept from Chapter 32 also requires dedicated views for value inspection because there are no result cells.

39 The Safe-DS Approach

Goals. In this thesis part, we want to develop an IDE for the Safe-DS language. It should

1. facilitate textual development in the Safe-DS language (Section 39.1),
2. offer a graphical view for creating pipelines (Section 39.2),

3. allow a seamless, bidirectional transition between the textual and graphical views for any valid pipeline (see below), and
4. provide views to display the results of pipeline runs (Section 39.3).

All these features aim to improve *safety*, i.e. prevent or detect mistakes and help recover from them, and should be as *user-friendly* as possible. Overall, our goal is to *accelerate development* of DS pipelines.

Goal 1. We already explained the textual Safe-DS language in Part III. It encompasses a pipeline language to create DS pipelines (Chapter 23) and a stub language to safely integrate Python libraries (Chapter 24). Both languages benefit from IDE support, which will be covered in depth in Section 39.1.

Goal 2. Since stubs can be created largely automatically (Section 24.4) and we expect library integrators to have some programming background, we decided not to create a graphical equivalent to the stub language. The graphical view corresponds solely to the simple pipeline language. This makes the graphical view simple, too, while still being expressive enough for pipeline development. The graphical view is outlined in Section 39.2.

Goal 3. We want users to be able to switch between the different views for any valid pipeline, while still profiting from static checks, like type checks (Section 25.4) or constraint checks (Section 25.5) in all views. This poses additional questions, though, especially if we consider possible future additions of views (whatever they might be):

1. How can we synchronize the different views while preventing combinatorial explosion regarding transitions between them?
2. How can we avoid implementing static checks for each view?

The solution to both issues is the *metamodel* (or *abstract syntax*) of the Safe-DS language. The metamodel is an abstract representation of language concepts, independent of a particular *concrete syntax*, such as certain text. An example is the *call*, which is characterized by a callee (the expression to call), and a list of arguments (the expressions to assign to parameters). In Chapter 23, we introduced the abstract concepts of the pipeline language alongside their concrete textual syntax for ease of reading. Individual DS pipelines are instances of the metamodel and are represented by an *abstract syntax tree* (*AST*).

Each view only needs to define a transition to and from the metamodel, which also makes the future inclusion of new views easy. Moreover, we can define the static checks once for the metamodel and each view just needs to decide how to display the detected problems. Figure 47 shows this principle from the perspective of a fixed pipeline. We can transition between a text and graphical view of the pipeline via the AST.

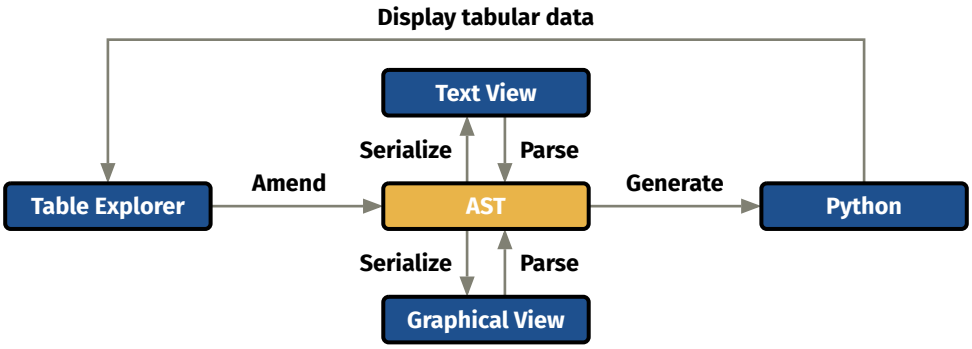


Figure 47: The abstract syntax tree (AST, yellow) connects views onto a pipeline (blue) via unidirectional or bidirectional links. Tabular data that is computed by running the generated Python code is presented in the table explorer. Static checks are implemented once using the AST.

The downside of this approach is the omission of details like code formatting or graph layout in the AST. These details could be stored alongside the AST. Since they are not relevant for program semantics, however, we instead decided to use an automated formatter for the code and an automated layout computer for the graph when serializing the AST to each view.

Goal 4. The Safe-DS library (Part II), which is integrated into the Safe-DS language, currently has containers for tabular data and images. Hence, the IDE needs ways to display tables, images, as well as primitive results like numbers (e.g. for the accuracy of a classifier). Since a Safe-DS pipeline is not written in a notebook, there are no result cells, which could display values in a generic manner. Instead, we offer dedicated views for the different types of data.

Primitive values get printed to a log. Single images are shown in a basic image viewer. `ImageList` objects cannot be displayed fully yet, but it is possible to inspect the individual images contained therein one-by-one with the image viewer. For tables, we developed a dedicated table explorer, which can display, sort and filter the data, show statistics, alert users of low data quality, and create plots. We discuss this further in Section 39.3.

To compute the values to display, the AST of a Safe-DS pipeline gets traversed to generate Python code, which is then executed by the runner (Chapter 32). The value of the requested placeholder is sent back to the IDE and shown in the appropriate view, like the table explorer. The table explorer manipulates the AST and executes the altered pipeline to provide its features, such as the creation of plots. This principle is also summarized in Figure 47.

39.1 Textual IDE

Language-agnostic features. The textual Safe-DS language is the core of our system and enables the use of version control systems like Git, or programming aids like GitHub Copilot[136]. As discussed in Section 38.1, IDEs already incorporate many such tools by default, as well as other language-agnostic features. These can be used out-of-the-box to edit Safe-DS pipelines and stubs.

A language server for Safe-DS. However, comfortable development requires language-specific features, too. For IDE independence, we provide a language server[137] for the Language Server Protocol (LSP), which handles all functionality that we present in this section unless otherwise specified.

We integrate this language server into a VS Code extension, so it can be quickly installed via the marketplace[138]. Third parties can create extensions for other IDEs, like PyCharm, with little effort by using our language server. The screenshots in this section are taken in VS Code, which provides the GUI elements to interact with the language server.

Diagnostics. The code in the editor is continuously statically checked to catch errors as early as possible. Faulty code is precisely marked by squiggly lines. Error messages are shown when hovering over the faulty code and include a potential resolution, where possible (Figure 48).

The language server checks for syntax errors, linking errors (unresolved references), type errors (Section 25.4), constraint violations (Section 25.5), and more. Additionally, it shows warnings if a bug is suspected, but the pipeline is still valid, e.g. if a placeholder is defined but never used. Finally, various informational diagnostics are provided, for example if syntax could be shortened. These informational diagnostics can be disabled in the settings.

Code-completion. Code-completion shows the names of all declarations that can be referenced in the current context. For example, the expression `titanic` in Figure 49 is a reference to a placeholder

[136] Once the workspace contains some Safe-DS code, the suggestions of GitHub Copilot are decent already. Further advancements would require more widespread adoption of the Safe-DS language.

[137] https://www.npmjs.com/package/@safe-ds/lang

[138] https://marketplace.visualstudio.com/items?itemName=Safe-DS.safe-ds

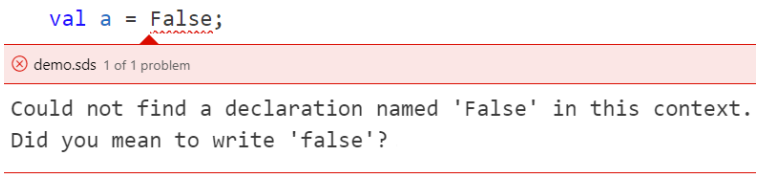


Figure 48: Diagnostics are shown directly in the code as it is being written. Markers are localized precisely to the faulty code. Messages provide potential resolutions, where possible.

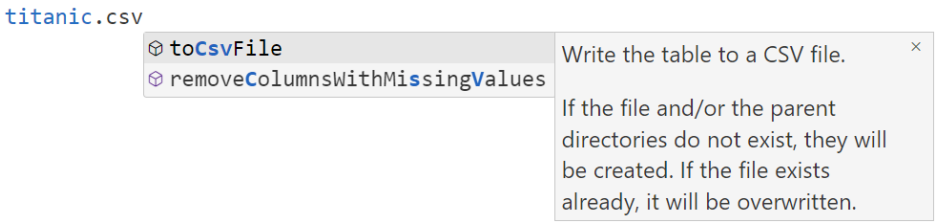


Figure 49: Code-completion shows all API elements that can be accessed in the current context (here the members of the `Table` class). Entering text filters the suggestions using fuzzy matching. The description for the selected entry is shown in an additional panel.

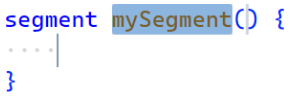


Figure 50: Snippets insert interactive templates into the code. They contain filler texts that should be changed by users (here the name, arguments, and body of the segment). Users can navigate between the filler texts with the tab key.

and has the inferred type `Table`. Hence, after typing a dot (member access), code-completion suggests instance members of the `Table` class. If the selected declaration has a documentation comment (Section 24.3), the description contained therein is rendered in an additional panel.

When users enter more characters, the suggestions are filtered by a fuzzy matcher. The first entered character (`c` in Figure 49) must match the beginning of a word in the declaration name, and all later characters must occur somewhere later in the name in the same order as well.

Moreover, the VS Code extension provides a list of snippets to help teach the syntax of the language[139]. These are included as additional entries in the code-completion dialog. If a user accepts a suggested snippet, an interactive template is inserted into the code. It contains filler text in locations where we cannot guess the user's intention, e.g. for names. The tab key allows navigation between these filler texts. Figure 50 shows the snippet for segment declarations, with filler texts for its name, arguments, and body. The name is currently selected, so users can start typing immediately to overwrite it.

Semantic highlighting. Basic syntax highlighting is defined using a context-free grammar[140]. This enables

- marking literals (e.g. `"titanic.csv"`),
- marking keywords (e.g. *`val`*),
- marking names of declarations (e.g. *`val`* `titanic`),
- using different styles for names depending on the kind of declaration (e.g. *`val`* `titanic` vs. *`class`* `Table`),

[139]The LSP does not yet support snippets, so this feature is currently tied to VS Code. The upcoming version 3.18 of the LSP can handle snippets [241].

[140]This is also how we implemented the syntax highlighting of Safe-DS code in this document.

```
val titanic = Table.fromCsvFile("titanic.csv");
val _histogram = titanic.plot.histograms();
val _rowCount = titanic.rowCount;
```

```
val titanic = Table.fromCsvFile("titanic.csv");
val _histogram = titanic.plot.histograms();
val _rowCount = titanic.rowCount;
```

Figure 51: Basic syntax highlighting vs. semantic highlighting. The latter is run after reference resolution, so it can style e.g. references to methods differently from references to classes.

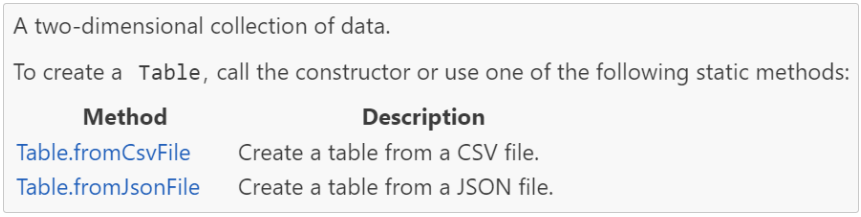


Figure 52: Tooltips are shown when hovering over declarations and references and render the corresponding documentation comment.

```
val titanic: Table = Table.fromCsvFile(path = "titanic.csv");
val _histogram: Image = titanic.plot.histograms();
val _rowCount: Int = titanic.rowCount;
```

Figure 53: Inlay hints show inferred information inside the code, like the type of a placeholder, or the name of the parameter that corresponds to a positional argument.

and more. Such basic syntax highlighting is included in the Safe-DS extension for VS Code. However, this context-free syntax highlighting is incapable of styling references to declarations (e.g. `titanic` and `Table`) the same way as the names of the referenced declarations: Since the grammars are not expressive enough to resolve references, we simply do not know their target.

The LSP instead offers *semantic highlighting*, which is run after reference resolution. Since we now know the target declaration of each reference, we can style the references as desired. Figure 51 shows the difference between basic syntax highlighting (left) and semantic highlighting (right).

Tooltips. Hovering over a declaration or reference shows the associated documentation comment (Section 24.1) in a tooltip (Figure 52). All Markdown they contain gets rendered, e.g. `_italic_` gets displayed as *italic*. Inline `{@link}` tags are replaced by hyperlinks to the corresponding declaration. Examples are presented with basic syntax highlighting[141].

Inlay hints. Inlay hints present inferred information to the user. They are shown interleaved with the code, but are not physically part of it (Figure 53). We use them to show the inferred types of placeholders and lambda parameters, as well as the name of the parameter that corresponds to a positional argument (e.g. for the `path` argument of `fromCsvFile` in Figure 53).

Users can enable or disable inlay hints for inferred types for placeholders and for lambda parameters independently in the settings. For parameter names, they can either opt to always or never show them, or decide to only show them if the argument value is a literal or if the value is not a simple reference.

Outline. The outline compactly displays the main symbols, like packages, pipelines, segments, and placeholders in a file (Figure 54). Clicking on an entry jumps to the corresponding location in the code. If a symbol is affected by diagnostics, like errors or warnings, it gets highlighted. The signature of segments (and callables in the stub language) is included, too. VS Code also offers a prompt to search for symbols in the opened file or in the entire workspace.

[141] Semantic highlighting in tooltips is not supported by the LSP.

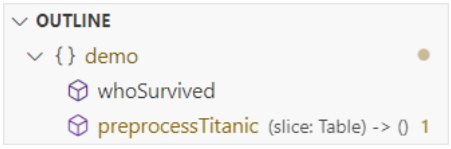


Figure 54: The outline view shows the main symbols in the current file, like the package (`demo`), pipelines (`whoSurvived`), and segments (`preprocessTitanic`). Symbols that contain diagnostics are marked (e.g. `preprocessTitanic`, which has one warning).

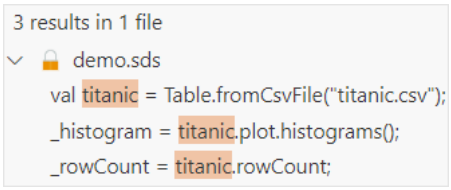


Figure 55: Searching for references to a symbol reveals its declaration and all places that use it, which works across files. Clicking on an entry jumps to the location in the code.

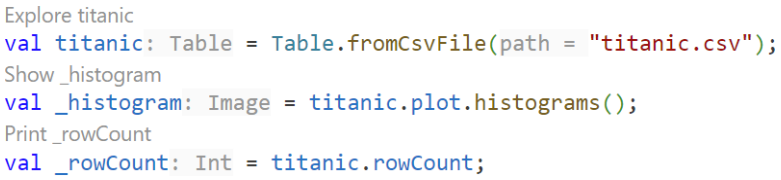


Figure 56: Values of placeholders can be inspected via *code lenses*, which are clickable actions in the code. Code lenses are displayed automatically based on the inferred type of a placeholder. They trigger partial execution (Section 31.2) and open an appropriate view for computed value.

Rename refactoring. Declarations are referenced by name. If the name of a declaration changes, all references must be updated, too. The rename refactoring helps with this change: It can be triggered from a declaration or reference and changes the name of the declaration and all references to it at once. This works across all files in the workspace.

Go to declaration/references. Similarly, users can go from a reference to the referenced declaration or show all references to a declaration. The latter is shown in Figure 55 and works across all files in the workspace. Clicking on an entry jumps to the location in the code.

Call graphs. Calls are special kinds of references, as they hand control to another program element. The LSP and VS Code provide dedicated presentations for call graphs, which can show incoming and outgoing calls for a callable, such as a segment. In our case, this helps users determine where impurity reasons originate from (Section 32.2).

Code lenses. Finally, we use *code lenses* to trigger partial execution of a pipeline, which are clickable actions in the code[142]. They either display the value of a placeholder or run the entire pipeline for its side effects (Section 31.2). The former is illustrated by Figure 56: The `"Explore titanic"` code lens opens the table in a dedicated table explorer (Section 39.3). The `"Show _histograms"` code lens displays the plot in an image view. The `"Print _rowCount"` code lens prints the result to a log. We decide which code lens to include based on the inferred type of a placeholder.

39.2 Graphical View

A graphical view for DS pipelines might be less daunting for non-programmers than a textual view. It also has the added benefit that certain categories of errors, like syntax errors, can be avoided, which

[142]Inlay hints are similar, but they are purely informational and have no associated action.

increases safety. Our graphical view displays a pipeline as a graph, with nodes for operations and directed edges for the data-flow between them. This corresponds closely to the fine-grained dependency graphs that we base our execution concept on (Section 31.1). Thus, the graphical view also helps users understand how operations are connected and how programs are executed.

To offer a bidirectional transition between the textual and graphical view, the graphical view must be able to express all abstract language concepts that compose the metamodel of the Safe-DS pipeline language. Only then can we define serialize and parse steps for the graphical view[143] (Figure 47). The metamodel of the pipeline language is small and encompasses expressions, statements, pipelines, and segments[144]. We now gradually introduce the graphical syntax for each concept and then explain how the graph can be edited.

39.2.1 Syntax

Generic expressions. Expressions are the most complex part of the pipeline language. They can also be expressed in a considerably more compact manner as text than as a graph. For example, splitting the expression `1 + 2 * 3` into separate nodes, would require five nodes, three for the integer literals, and two for the arithmetic operations.

Thus, we first define a generic expression node (Figure 57), which acts as a fallback if an expression kind has no specialized representation. It has a single *output port*, which is the small circle on its right side. An edge that originates from the output port indicates an access to the value of the expression.

Only a single edge may be connected to the output port. To use the value of an expression multiple times, it must be assigned to a placeholder (see below). A generic expression node covers the largest possible expression, i.e. one node is created for the `1 + 2 * 3` example above. The node remembers the corresponding text, which is used for the parsing step.

Calls. A graph that only shows generic expression nodes would be useless. Displaying calls is of particular importance, as they define the core semantics of a program.

We follow an approach similar to Altair AI Studio and other graphical tools and show calls as nodes with *input ports* for parameters (left side) and *output ports* for results (right side). Required parameters have a white port, optional ones a gray port. The shape of the node is fully determined by the signature of the called function (Figure 58). If a required parameter is not set yet, its port has a red border to alert users of this error.

Figure 57: A generic expression node for any expression without dedicated graphical syntax. Its value is accessed by an output port, which is the small circle on the right.

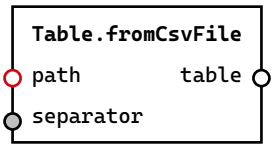


Figure 58: A call node with input ports on the left for parameters and output ports on the right for results. Input ports are filled with white for required parameters and with gray for optional one. Ports for required parameters that are not set have a red border to highlight this error.

[143] Note that the parse step for the text view is undefined for programs with syntax errors, so switching from the text view to the graphical view is only possible for programs without syntax errors.

[144] This is a major advantage over a general-purpose language like Python, where a graphical equivalent would also need to encompass concepts like conditionals, loops, etc.

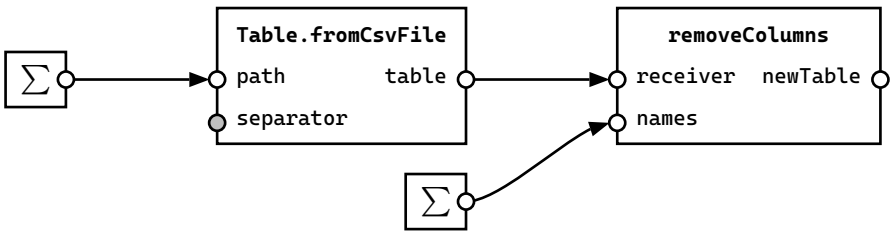


Figure 59: Two chained method calls. Edges going into the input ports pass arguments. Edges coming out of output ports access that result. The receiver is modeled as another input port.

The addition of call nodes already facilitates expressing arbitrary nesting or chaining of calls. For example, the code

```
1 Table
2     .fromCsvFile("titanic.csv")
3     .removeColumns("id")
```

is represented by the graph in Figure 59. The receiver of a method call is represented by another input port and is always required (i.e. white). An edge that ends at an input port passes an argument. An edge that starts at a output port accesses that result. Only a single edge may be connected to input ports, since we may not define multiple arguments for the same parameter in a call. Likewise, only a single edge may be connected to output ports; reusing the result necessitates assigning it a placeholder (see below).

Sidebar for primitive parameters. Representing each literal argument by a generic expression node quickly clutters the graph (Figure 59), while providing little additional information to the user. Like Altair AI Studio and other graphical tools, we solve this issue by offering a sidebar to enter literal values for primitive parameters (Figure 60).

The sidebar can be opened by clicking on a call node to select it. The selection is visualized by the blue border around the node for `fromCsvFile` in Figure 60. The sidebar shows suitable input fields for all primitive parameters of the callable (e.g. a number input field for integers). The input fields display the current values and allow users to change them. They also show the default value of optional parameters as a filler text until users overwrite it, like the comma for `separator`.

We still keep the ports for primitive parameters on the call node, as users might compute their values using function calls or other complex expressions. To indicate that an argument was entered in the sidebar, we fill the parameter node with black. Hovering over a black port displays the associated value. Connecting an edge to a port for a primitive parameter overwrites the value in the sidebar and disables the associated input field.

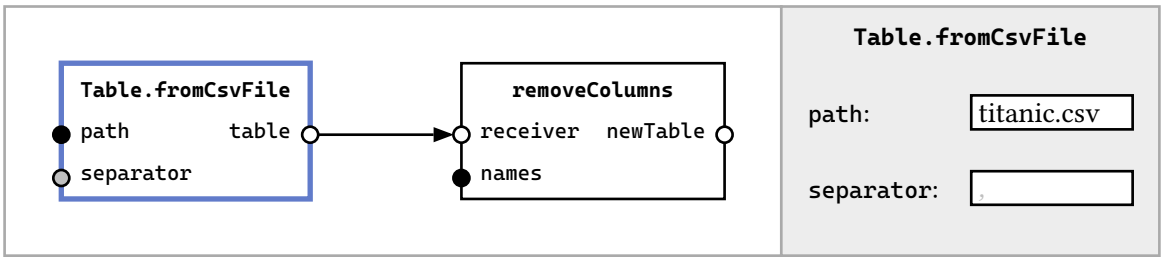


Figure 60: Literal arguments can be viewed and edited in a sidebar to reduce clutter in the graph. It is opened by clicking on a call node. Here, the node for `fromCsvFile` is selected (indicated by the blue border). Input ports are filled with black to denote that their value is entered in the sidebar.

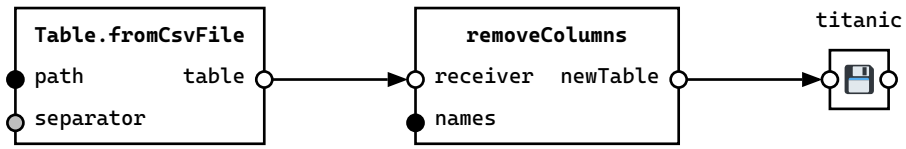


Figure 61: Assigning the result for the call to `removeColumns` to the placeholder `titanic`. Assignment is indicated by an edge that ends at the input port (left side) of a placeholder node. The value of a placeholder can be retrieved by the output port (right side) of the node.

Assignments. Besides expressions, we need graphical syntax for statements, particularly for assignments. Similar to the dependency graphs (Section 31.1), placeholders get their own node in the graph. The node has an input port (left) and an output port (right). Ending an edge at the input port assigns a value to the placeholder. Starting an edge at the output port accesses its value (Figure 61).

Since placeholders are immutable, only a single edge can be connected to the input port. The value can be accessed as often as needed, so multiple edges can be linked to the output port. If no edge ends at the input port, this indicates that the placeholder is unassigned. In this case, the input port gets a red border to emphasize this error.

Expression statements. Expression statements are a technical necessity caused by the textual Safe-DS language, as the body of a pipeline, segment, or block lambda cannot contain loose expressions[145]. But since they only evaluate the expression they contain, we do not add special nodes for them in the graph and display them identically to the expression they contain.

We can still recognize expression statements because they end with a generic expression node or call node that is a leave. Then we follow the data-flow edges backward until we arrive at placeholder nodes (these are not included) or root nodes (these are included). For example, we determine that all nodes in Figure 59 belong to a single expression statement and that Figure 61 contains no expression statements.

Pipelines. The meaning of a pipeline is determined by the statements in its body. Hence, the visualization for a pipeline is equivalent to the combined visualization for its statements (and showing the name of the pipeline).

Segments. When a segment is called, we show a normal call node, whose shape is determined by the signature of the segment. Unlike for functions in the stub language, we also know the implementation of a segment, i.e. the statements in its body. These statements can be viewed by clicking on the call node. This zooms into the node and shows how parameters are used internally and how results get computed. Figure 62 shows the visualization for the segment

```
segment loadData(path: String) -> table: Table {
    yield table = Table
        .fromCsvFile(path)
        .removeColumns("id");
}
```

In the internal view, an input port is shown for each parameter and an output port for each result. The port for `path` is not filled because the internal view represents the segment declaration, and not a specific call of it. An edge that originates from an input port indicates a reference to this parameter. An edge that ends at an output port indicates an assignment (*`yield`*) to that result.

Multiple edges can be connected to the same input port, as a parameter may be referenced more than once. Only a single edge may be connected to each output port, as we can only set a result once. If a result port has no incoming edge, i.e. it is unassigned, it is marked with a red border to mark this error.

[145]The semicolon at the end of an expression statement unambiguously denotes its end.

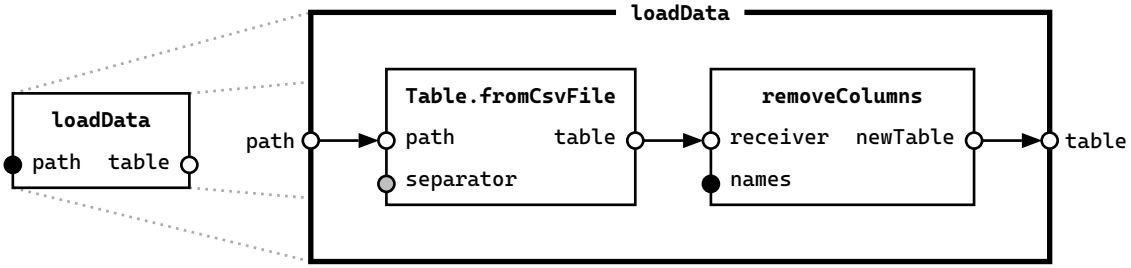


Figure 62: External view of a segment call and internal view of its implementation. In the internal view, parameters and results are still shown by input ports (left) and output ports (right) respectively. But now, edges *start* at input ports (parameter reference) and *end* at output ports (`yield` of result).

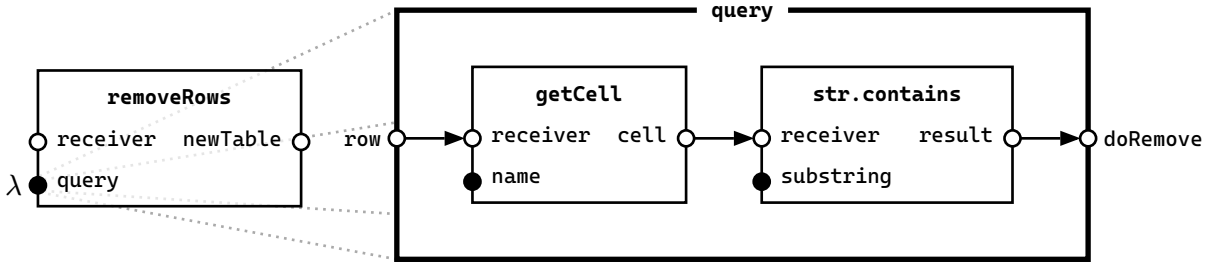


Figure 63: Lambdas get inlined into the input port for callbacks of higher-order functions. Clicking on the port reveals the implementation of the lambda.

Lambdas. Lastly, we can improve the representation of lambdas. A lambda is an expression, so it would currently be visualized by a generic expression node. But similar to pipelines and segments, lambdas have a body[146] that we would like to zoom into. This interaction requires special treatment.

Since lambdas must only be passed as arguments to named callables and callbacks must be specified by a lambda, we do not need lambda nodes in the graph and can inline them into the input port of a call node. Clicking on the input port reveals the implementation of the lambda. Figure 63 shows this for the call

```
table.removeRows(
    (row) -> row
        .getCell("name")
        .str.contains("Smith")
)
```

For brevity, the data-flow edge into the `receiver` port of the call node for `removeRows` is omitted. The input and output ports of the internal lambda view, as well as the associated names, are known from the type of the callback parameter (Section 25.1). Data-flow edges in the internal view for lambdas work the same as for segments.

39.2.2 Editing and Value Inspection

Editing. We already discussed the sidebar in Section 39.2.1, where users can set primitive parameters to literal values. To assign the result of a complex expression to a parameter, users must add generic expression nodes and call nodes to the graph. The expression for generic expression nodes is entered as text. As discussed before, building the expression out of individual nodes would be impractical.

Generic expression nodes and call nodes are selected from a toolbox (Figure 64), dragged onto the graph, and then connected with edges between their ports. Related entries in the toolbox are grouped based on the argument of the associated `@Category` annotation (Listing 33). Entries can be filtered by

[146]For block lambdas, the body is a list of statements; for expression lambdas, it is a single expression (Section 23.1).

Figure 64: A toolbox to add nodes to the graph. Entries are grouped by category and can be searched. Selecting an output port in the graph keeps only the entries with compatible input ports.

```
class Table {
    @Category(PipelineStep.Import)
    static fun fromCsvFile(path: String, separator: String) -> table: Table

    @Category(PipelineStep.Export)
    fun toCsvFile(path: String)
}
```

Listing 33: The `@Category` annotation accepts a variant of the `PipelineStep` enumeration. This information is used to group related entries in the toolbox together.

a textual search, or by selecting an output port in the graph. In the latter case, only operations that have an input port of a compatible type are shown. Adding such a node to the graph automatically connects the selected output port to the matching input port of the new node if there is no ambiguity.

Value inspection. Values of placeholders in a pipeline can be inspected via a context-menu. The context-menu functions identically to the code lenses from the textual IDE, i.e. its entries are determined by the inferred type of the corresponding placeholder: Primitive values can be printed, images are displayed, and tables are opened in the table explorer (Section 39.3).

39.3 Table Explorer

The table explorer is shown when users inspect the value of a placeholder of type `Table`, e.g. by clicking on the associated code lens (Section 39.1). It combines features from various state-of-the-art tools (Section 38.3) and helps users quickly understand their data without writing code or altering the graph.

Main panel. The main panel is an integrated view for the data contained in a table, statistics or visualizations for each column, as well as checks regarding data quality (Figure 65[147]). We refer to the part that compiles statistics, visualization, and checks as the *profiling section*.

Profiling. The column type and value distribution determine which statistics or visualizations are shown:

- The `name` column contains text and has many distinct values, so we only show basic information.
- The `sex` column also contains text, but has only two values, so we show their distribution.
- The `age` column contains numbers, so we show a histogram.

For all columns, we check whether they contain any missing values. These must be imputed or otherwise handled before the column can be used as a feature to train one of our ML models. To alert users of this issue, the corresponding text is written in red and an error icon is shown next to the "Hide/Show Profiling" toggle. The icon is visible even when the profiling section is hidden. The profiling section is designed to be easily extensible.

[147] All figures in this chapter are screenshots from the implementation of our concept. The implementation was done by Bela Bothin as part of his Bachelor's thesis [242].

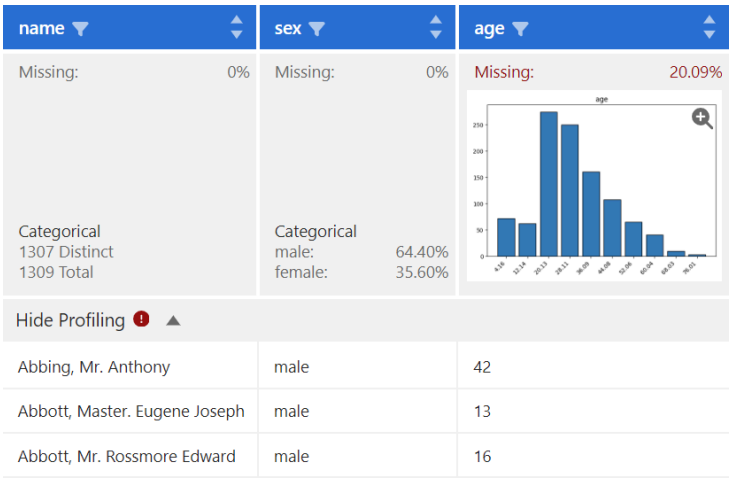


Figure 65: The main panel of the table explorer shows the data as well as statistics, visualizations, and quality metrics for each column. Potential quality issues are highlighted in red.

Hiding, sorting and filtering. To reduce clutter, a column can be hidden from view via an entry its context-menu. Columns can also be sorted in ascending or descending order by clicking on the arrows in the blue column header. Additionally, rows can be filtered by clicking on the funnel icons. This opens different dialogs depending on the type and distribution of the column. Users can filter `name` using text-based filters, `sex` by selecting options from a dropdown, and `age` by configuring a range. The latter is shown in Figure 66.

Filtering invalidates the profiling section, so after changing filters, we display a button to refresh it. Refreshing is not done automatically, since recomputing the profiling information can be expensive for large datasets and users might want to configure multiple filters before updating the profiling section.

Creating plots. Moreover, the table explorer has a dedicated plot editor (Figure 67). Here, users select a plot type, and the columns to visualize. We currently provide box plots and histograms for a single column, line or scatter plots for a column pair, and correlation heatmaps for the entire table, but the plot editor is designed with future extensions in mind.

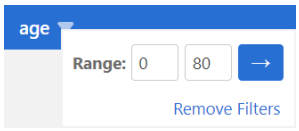


Figure 66: Clicking on the funnel icon opens a filter dialog. Its input fields are determined by the type and distribution of the column. Since `age` is numeric, users filter values using a range.

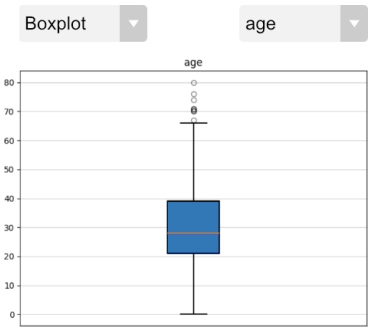


Figure 67: In the plot editor, users pick a plot type and the columns to visualize in dropdowns. For this figure, the actual plot has been shrunk in relation to the dropdowns to save space.

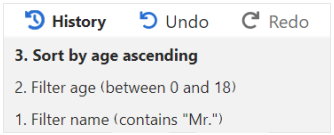


Figure 68: The history shows all actions done in the table explorer that change data. The current step is printed in bold, and steps can be undone or redone.

There are several ways to open and use the plot editor:

- It can be opened in a blank state and configured by dropdowns.
- It can be opened by clicking on a visualization in the profiling section (magnifier icon in Figure 65). This opens an enlarged version of this plot.
- It can be opened by selecting columns and choosing the plot in the context-menu. Which plots appear in the context-menu depends on whether no, one or two columns are selected.

Action history. We refer to any computation in the table explorer as an *action*. Some actions happen automatically, e.g. getting the initial profiling information. Other actions are triggered by the user, such as creating plots, and sorting or filtering the table. The latter two are currently the only actions that change data. The table explorer keeps a history of all actions that change data, so users can undo and redo them (Figure 68). The current state is marked in bold text.

Executing actions. We differentiate actions based on how they are executed: *Internal actions* only affect the presentation of the data and are handled completely inside the table explorer, e.g. to hide/show a column. *External actions* require computations that are potentially costly, e.g. filtering the data, calculating statistics, and creating plots. The Safe-DS library already implements these operations efficiently, so the table explorer

1. augments the AST of the existing Safe-DS pipeline,
2. generates Python code (Section 32.3),
3. executes the Python program with the runner (Section 32.4), and
4. displays its results.

This enables full use of the optimizations provided by our execution approach, i.e. the computation of a backward slice (Section 31.2) and memoization (Section 31.3). When running a pipeline later, all results computed by the table explorer are already available. Likewise, if a previous pipeline run already computed values that are needed by the table explorer, they are immediately retrieved from the memoization table.

40 Implementation (IDE)

Acknowledgments. The table explorer was developed by Bela Bothin as part of his Bachelor's thesis [242]. The implementation of the graphical view (without editing capabilities) is the topic of the ongoing Bachelor's thesis of Gideon König [243].

Progress and repositories. The implementation of the textual IDE (Section 39.1) and table explorer (Section 39.3) is complete. The current implementation of the graphical view (Section 39.2) supports the syntax to visualize pipelines, the sidebar for primitive parameters, and the toolbox for generic expression nodes and call nodes. However, neither editing values in the sidebar nor the modification of the nodes and edges in the graph is possible yet, i.e. a pipeline can only be displayed but not changed. All features are integrated into the open-source repository for the Safe-DS language[148].

[148] https://github.com/Safe-DS/DSL

Textual IDE. As explained in Chapter 26, the language is implemented in TypeScript and based on the Langium framework [174], which derives a set of services from the specification of the grammar and metamodel of a language. Besides easing the development of command-line tooling for a language, Langium also facilitates the implementation of language-specific IDE features by providing additional services and integrating them into the Language Server Protocol (LSP).

The default behavior of these services is rarely *exactly* what is required, and some services do nothing at all by default from the user's perspective. However, all services take care of the low-level interaction with the requests, responses, and notifications of the LSP and hide it from the developer. Instead, Langium provides clearly defined extension points to gradually change the default behavior.

For example, we specified when and how inlay hints should be displayed by writing and registering a subclass of `AbstractInlayHintProvider`, which only needs to override the method `computeInlayHint` to compute the inlay hint for a given AST node. Langium handles the interaction with the LSP, parsing of the document in the editor, traversal of the AST, calling this method, and collecting results.

Graphical view and table explorer. The graphical view and table explorer are only available in the VS Code extension for the Safe-DS language, as they mainly contribute a GUI, which cannot be handled by the LSP. They are each provided as a separate Webview [244], which can essentially render arbitrary HTML. Each Webview is implemented with Svelte, which offers reactivity and an abstraction for components. We picked Svelte since it offers a suitable combination of performance, ease of development, and maturity. Figure 69 is a screenshot of the work-in-progress Webview for the graphical view.

41 Evaluation (IDE)

We focused our evaluation effort on the Safe-DS pipeline language and its IDE features, as this is the part most users will interact with. The stub language (Chapter 24) allowed us to safely integrate the full Safe-DS library (Part II), and we could create stubs largely automatically with the generator (Section 24.4). Only the purity information (Section 32.1) had to be updated manually, which was done easily in the textual IDE with complete language support. We did not yet evaluate the graphical view (Section 39.2), as its usability and usefulness can only be reasonably determined once it can edit pipelines.

This leaves four main aspects that we did evaluate:

- The textual pipeline language (Chapter 23).
- The textual IDE (Section 39.1).
- The execution concept for Safe-DS pipelines (Chapter 32), including code lenses to start execution.
- The table explorer (Section 39.3), and other views for inspecting results.

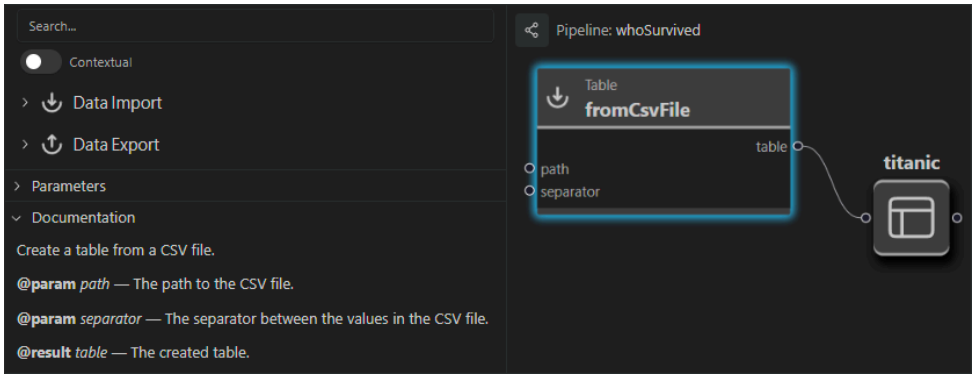


Figure 69: A screenshot of the work-in-progress implementation of the graphical view.

These aspects are inseparable from a user's perspective to develop DS pipelines, so we evaluated them together with three different methods:

- Computer science students used the pipeline language over multiple sessions to find potential issues (Section 41.1). All feedback was immediately acted upon, so the language evolved continuously.
- Medical students used the pipeline language during a workshop (Section 41.2). Here, we wanted to assess how users without DS knowledge and little to no prior programming experience would cope.
- A structured usability study compared the usability and development speed of the Safe-DS language and IDE to the Safe-DS library (Section 41.3). Here, we used fixed versions for consistency.

41.1 Bachelor Lab

We already explained in Section 18.1, how eight students used the Safe-DS *library* to solve Kaggle tasks during six sessions of the Bachelor lab "Angewandte Softwaretechnologien 2024". Another separate group of eight students worked on the same tasks in parallel but used the Safe-DS *pipeline language* and its IDE.

Feedback was immediately acted upon between the sessions, e.g. by extending the Safe-DS library and updating the stubs, clarifying error messages, or otherwise improving the IDE. The IDE features presented in this thesis part already incorporate these modifications. The pipeline language itself remained constant during this experiment, as we received no requests for changes.

41.2 Medical Workshop

In November 2024, we were invited to a workshop organized by the Institut für Digitale Medizin[149] of the University Hospital Bonn. The goal of the workshop was to teach medical students the very basics of programming and data analysis. In previous iterations of the workshop, this was done solely with Python, pandas, Matplotlib, and Jupyter Notebook.

Participants & groups. This time, one group of six students still used those tools (Group 1), while another six students worked with our pipeline language and IDE (Group 2). Participants were roughly separated into the two groups based on their own estimate of prior programming and data analysis experience. Both groups had three participants without experience, and three participants who studied computer science in school.

Supervision. Group 1 was supervised by three tutors, while Group 2 was supervised by one tutor (Lars Reimann). As the main goal of the workshop was teaching students, all questions were directly answered, though the tutors did not provide complete solutions to tasks. Participants were allowed to help one another.

Tasks. Both groups had 1:45h to finish a series of tasks on a small dataset with four columns. The tasks were the same as in the previous iterations of the workshop, and covered

1. loading the data,
2. determining the number of rows and columns in the table (Group 2 did this once in the table explorer and once with code),
3. basic table manipulations, like keeping only the first five rows,
4. filtering rows (Group 2 had to use higher-order functions for this), and
5. visualizing the data with plots.

For the final task, which involved more complex queries to filter the data, Group 2 was allowed to either write code or use the table explorer. All tasks for both groups included detailed instructions whenever new language concepts or API elements were needed for the first time. When a known language concept or API element was required later, participants had to figure this out themselves.

[149] https://www.idm.uni-bonn.de

Result: Finished tasks & time required. Out of the six participants in Group 1, only one was able to finish all tasks and needed the entire allotted time (1:45h). In each previous iteration, just a few participants could solve all the tasks in time, too. In Group 2, four participants finished all tasks and took between 1:10h and 1:40h. The other two participants had to leave early, unfortunately, but they still managed to finish all but the final task after 1:14h and 1:24h and likely would have finished it in the remaining time.

It is noteworthy that all participants in Group 2 opted to write code to solve the final task rather than use the table explorer, which likely would have been quicker still. When asked for the reason afterward, the remaining four participants agreed that they wanted to practice programming. One participant stated that he thought using the table explorer would be "too fast and simple".

Result: Usability. All participants filled in a System Usability Scale (SUS) questionnaire [69], [70] after finishing the task, depleting the 1:45h, or before they had to leave, whatever came earliest. Groups 1 and 2 gave a mean SUS score of 59.2 and 71.3, and a median SUS score of 56.3 and 72.5 respectively. On the adjective scale [72], [73], the mean score from Group 1 is *OK*, while the mean score of Group 2 is at the boundary between *OK* and *good*.

50% of the SUS scores analyzed in [72], [73] were below 70.5, so the score from Group 2 is in the upper half. Note that the SUS score is mostly used to determine the usability of less technical systems, like full GUIs or webpages, so we reckon that this result is great for a programming language, especially from participants with little to no programming experience.

41.3 Structured Usability Study

Acknowledgments. The usability study was conducted by Camilla Grefrath as part of her Bachelor's thesis [140].

To rule out any potential bias in the workshop results (Section 41.2) regarding the different tutors, or possible group dynamics, we carried out a structured usability study. Here, participants solved a task with the help of a moderator in one-on-one sessions. The moderator provided fixed hints if a participant could not progress on their own.

The details about the study setup were already explained in Section 18.2. To summarize, seven computer science Bachelor students worked on the Titanic dataset with some original Python libraries for DS (Group 1), and another seven students with the Safe-DS library (Group 2). Both groups worked with notebooks in VS Code and extensions for Python language support were installed. Another seven students (Group 3) solved the task with the pipeline language in VS Code with the Safe-DS extension installed (v0.22.0). The level of prior experience was similar for all groups, and nobody had worked with or even heard of any Safe-DS tool before.

By using the same IDE (VS Code) and library (Safe-DS) between Groups 2 and 3, we ensured that only the parameters differed that we were interested in, i.e. the language, language-specific IDE features, execution concept, and value inspection. This allows us to answer the following research questions, where *full Safe-DS stack* is meant to subsume the pipeline language, the features of the Safe-DS extension for VS Code, our execution concept, and the table explorer and other means to view results:

- **RQ1**: How does the *general usability* of the full Safe-DS stack compare to the Safe-DS library?
- **RQ2**: How does the *learnability* of the full Safe-DS stack compare to the Safe-DS library?
- **RQ3**: Can users of the full Safe-DS stack *finish more tasks alone* compared to the Safe-DS library?
- **RQ4**: How does the *development speed* with the full Safe-DS stack compare to the Safe-DS library?

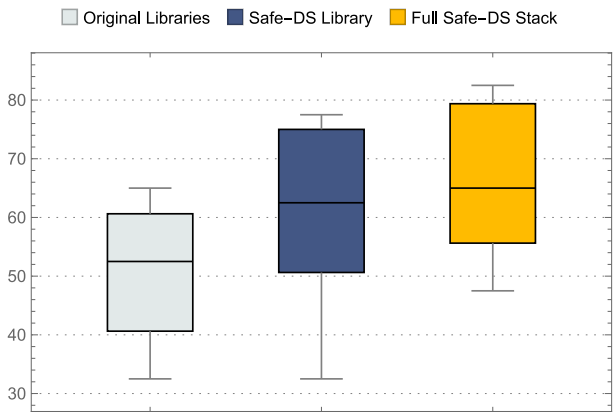


Figure 70: A box plot of the SUS scores for all groups. Higher values are better.

Result (RQ1): SUS scores. Participants filled in one SUS questionnaire [69], [70] about the usability of the Safe-DS library in Group 2 and the full Safe-DS stack in Group 3.

The Safe-DS library got scores between 32.5 and 77.5, with a mean of 60.7, standard deviation of 16.3 and median of 62.5. The mean score corresponds to *OK* usability on the adjective scale [72], [73]. The full Safe-DS stack got scores between 47.5 and 82.5, with a mean of 67.1, standard deviation of 13.7, and median of 65. On the adjective scale, this mean score corresponds to *OK* usability. Figure 70 shows the distributions in a box plot.

Result (RQ2): SUS-like scores for learnability. As explained in Section 18.2, the questionnaire contained one custom item "It took me a long time to learn how to use the system." at the end. We used this item in addition to Items 4 and 10 of the standard SUS questionnaire to assess the learnability of the tools used in each group. The ratings for these items were normalized to a score between 0 and 100 (higher is better).

The Safe-DS library had scores between 8.3 and 75, with a mean of 39.3, a standard deviation of 22.4 and a median of 41.7. The full Safe-DS stack received scores between 25 and 100, with a mean of 67.9, standard deviation of 23.8, and median of 75. There is not enough data to compare this result to other systems. The value distributions are summarized in Figure 71.

Result (RQ3): Required help. In Group 2 (Safe-DS library), one participant was unable to solve two steps on their own. In Group 3 (full Safe-DS stack), all participants could solve all steps on their own.

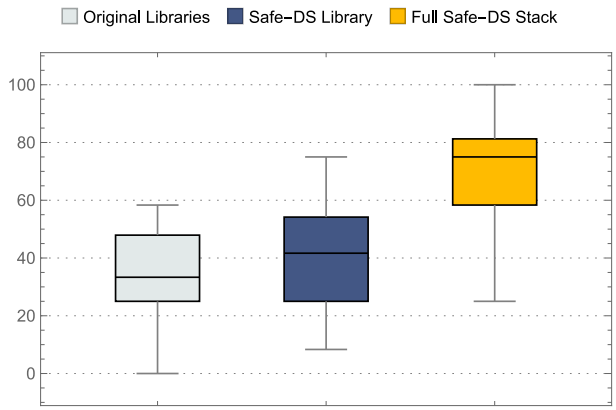


Figure 71: A box plot of the SUS-like scores for learnability for all groups. It is based on two items of the normal SUS questionnaire, and a custom one. The final score is between 0 and 100. Higher values are better.

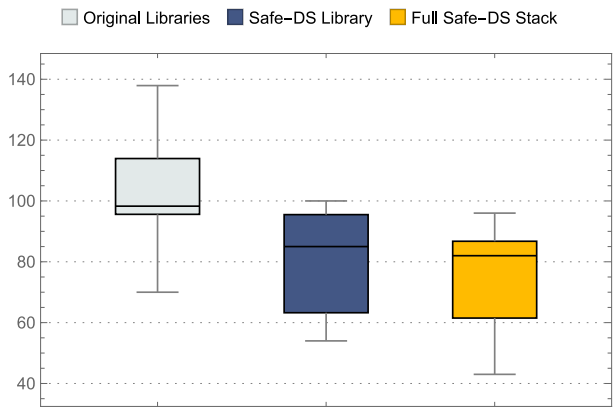


Figure 72: A box plot of the time (in minutes) to solve the task in all groups. If someone could not solve a step on their own, we used the mean time that other participants who did solve it required. Lower values are better.

Result (RQ4): Time for the task. Finally, we measured how long the participants took to solve the task. If a participant was unable to solve a step on their own, we substituted the *mean* time needed by the other participants from any group that did solve the step. Since this affected only the two steps from Group 2, the imputation rule has negligible effect.

The imputed times for the Safe-DS library were between 54 and 100 minutes, with a mean of 79.5 minutes, a standard deviation of 18.3 minutes, and a median of 85 minutes. For the full Safe-DS stack, the times were between 43 and 96 minutes, with a mean of 75.7, standard deviation of 19.2, and median of 82. Figure 72 shows both distributions as a box plot.

Summary. Compared to the Safe-DS library, which already improved upon all aspects compared to the original Python libraries, the full Safe-DS stack

- is more usable (plus 6.4 points in the mean SUS score),
- is much easier to learn (plus 28.6 points in the mean SUS-like learnability score),
- lets DS novices get more work done on their own (no incomplete steps), and
- enables faster development (minus 4.8% mean required time).

41.4 Discussion

The discussion of the threats to validity from Section 18.3 regarding lack of DS and VS Code knowledge, moderator bias, and expressiveness applies to Group 3, too. We additionally want to highlight that some participants of Groups 1 and 2 had already worked with Python or a notebook environment before, so they had some familiarity with the language and the cell concept for execution. In Group 3, *all* participants had to quickly learn from scratch

- the API of the Safe-DS library,
- the concepts and syntax of the pipeline language,
- the textual IDE features,
- the execution concept, and
- the views for value inspection, including the table explorer.

Despite that, the full Safe-DS stack received a far better SUS-like score for learnability.

Moreover, by design of the study, the developed pipeline could always be executed fully in a matter of seconds. Thus, the impact of runtime errors or an inefficient cell structure was minimal. Because of this, we could neither fully evaluate the impact of the continuous static checking, nor of our correct and minimal execution concept. The full Safe-DS stack still improved upon the Safe-DS library regarding

all metrics we analyzed. The outcome of this usability study also confirms the results we presented for the medical workshop (Section 41.2).

42 Future Work (IDE)

Editing in graphical view. The graphical view (Section 39.2) can already visualize pipelines and show dependencies between operations, which helps users understand how their pipeline behaves. However, editing functionality is missing at the moment. This must be added, so the graphical view becomes as useful as the text view. Note that the challenging direction, i.e. moving from text (high variability) to graph (low variability), is already done. Once we can edit pipelines in the graphical view, its usability can be assessed, too.

Data preparation with table explorer. The table explorer (Section 39.3) currently focuses on facilitating the understanding of data, e.g. via statistics or plots. Sorting and filtering are the only operations that change the data. Akin to Data Wrangler, we could offer more operations for data preparation. These are already contained in the Safe-DS library, so each could be realized as another external action and integrated into the table explorer via its context-menu or other GUI elements.

Saving the action history. The table explorer keeps a history of all executed actions. However, there is no way to save it, so if the IDE is closed, the action history is lost. This becomes a problem once we add more operations for data preparation.

There is a straightforward solution, though: The table explorer creates a modified Safe-DS pipeline in the background to execute its external actions. This pipeline contains an additional placeholder for the final state and method calls for all actions in the history. We could add an option to export this modified pipeline, so it can be serialized to text, versioned with Git, and easily shared with colleagues. Later, the table explorer can be reopened on the new placeholder.

One-way transitions to Safe-DS pipelines. Conceptually, the textual and graphical view onto a Safe-DS pipeline are equivalent, so we can transition between them. The enhanced table explorer described above only offers a one-way transition to a pipeline, i.e. we can use it to create or modify a pipeline but not to visualize an arbitrary pipeline.

Likewise, we could offer other one-way transitions to a Safe-DS pipeline, like a wizard or chat interface. Users could use this interface to create the initial Safe-DS pipeline and later tweak the result textually or graphically as needed.

More views for value inspection. Similar to the table explorer, we could implement a more sophisticated image viewer for individual images that also has editing capabilities. These operations are already included in the Safe-DS library, so only the GUI is required. Ideally, it would also be able to handle image lists, similar to photo organizers. Furthermore, if we add interactive plots to the library, we also need a way to show them in the IDE.

Debugger. The IDE offers no debugger for Safe-DS pipelines yet. But they are compiled to Python, which has a debugger. Since we create source maps, we can translate Safe-DS breakpoints to Python and the current execution state of the Python program back to Safe-DS. Thus, the future implementation of a debugger for Safe-DS is feasible.

43 Conclusion (IDE)

In this final thesis part, we introduced an integrated development environment (IDE) for the Safe-DS language. Language support for the textual pipeline and stub languages (Section 39.1) encompasses continuous static checking with helpful error messages, code-completion that filters suggestions based on the current context, tooltips to show documentation, inlay hints to show inferred information like types, and more.

All this functionality is available in a language server, so it can be integrated easily into modern editors and IDEs that support the Language Server Protocol. We also provide an extension for VS Code that bundles the language server. VS Code offers comprehensive language-agnostics features out-of-the-box, like a powerful text editor, and the integration of version control systems.

Beyond that, our VS Code extension includes a graphical view onto Safe-DS pipelines (Section 39.2). It can already display any syntactically valid pipeline. This cannot be done in a user-friendly manner for a general-purpose language like Python due to the amount of language concepts to cover. The comparatively simple addition of editing capabilities is future work. Then users will be able to switch between the text and graph view for any valid program, so they can always use the view best suited for their current task.

Pipelines can be partially executed by clicking on code lenses created by the language server, which triggers code generation (including the computation of a backward slice) and code execution with the runner (including memoization), as explained in Chapter 32. The computed results are shown in dedicated views in VS Code: Primitive values like strings are printed to a log, images are shown in a simple image viewer, and tables are opened in a custom table explorer.

The table explorer (Section 39.3) displays the data in the table, as well as statistics, quality metrics, and visualizations for all columns. The data can be sorted and filtered, and users can create additional plots in a simple dialog. Full integration of the table explorer into the same execution system we use for pipelines allows transparent reuse of results.

A usability study (Section 41.3) compared a typical development stack that consisted of Python, various DS libraries (Polars, Matplotlib, and scikit-learn), and a notebook environment to the full Safe-DS stack (library, language, execution concept, and IDE). It indicates that the full Safe-DS stack

- is considerably more usable (plus 16.4 points in the mean SUS score),
- is significantly easier to learn (plus 34.5 points in the mean SUS-like learnability score),
- lets DS novices get much more work done on their own (no vs. 40 incomplete steps out of 119), and
- greatly accelerates development (minus 26.3% mean required time with a generous imputation rule).

Hence, the full Safe-DS stack is a worthwhile alternative to the traditional Python stack for the development of DS pipelines.

Conclusion

This thesis introduced Safe-DS for the development of DS pipelines, and the generally applicable adaptoring process to create an alternative API for an original library:

Adaptoring (Part I). Many mature and well-tested libraries for DS exist already, particularly for Python. Developing another DS library from scratch and continuously maintaining it in the future is a major, and ultimately futile effort, since it is impossible to compete with the speed at which large development teams evolve these libraries.

However, the original DS libraries have huge APIs to cover most imaginable use cases, no matter how rarely they occur. Because of this, many API elements are hardly, if ever, used. For instance, we analyzed 41,867 programs mined from Kaggle and found that 21% of the classes and 47% of the functions of the public API of scikit-learn are unused. 58% of the parameters are unused or always set to the same value (Section 10.1). Including such API elements only makes it more difficult to discover the API elements that are needed to solve a given task.

Moreover, the design of several used API elements of the original DS libraries is not ideal. For example, 112 parameters of the public API of scikit-learn have strongly misleading default values (Section 10.1). Preconditions of functions are at best checked early at runtime and at worst only included in natural language in the documentation, so violations lead to failures late into a program run or subtle bugs.

Adaptoring is a means to provide a new, improved API for a library. The new API is implemented using the adapter pattern and calls the original library internally. Because no code is duplicated, any bug fixes or performance improvements to the original library can be consumed immediately. The original library is not changed, so adaptoring does not break clients of the original library and avoids the need to bloat the original API further by deprecations.

Many steps of the adaptoring approach are automated to make it viable even for large libraries: Deletions (Section 4.1 and Section 4.2) and parameter optionality (Section 4.3) are inferred from usage data gathered from representative client code. Several types of preconditions are extracted from the library's documentation (Chapter 5). Given a set of transformations, adapter code gets generated automatically as well (Chapter 6). Inferred transformations can be reviewed and edited in a specialized GUI (Chapter 7). An API designer can also add transformations that cannot be inferred via this GUI, including renaming and moving an API element.

The extraction of preconditions from documentation achieved high precision (88% to 92%) and varying recall (53% to 94%) on scikit-learn (Section 10.2). Recall can be improved by adding more text processing rules to cover missed cases. All automated steps together took less than 31 minutes on scikit-learn and 41,867 client programs (Section 10.3). The GUI for manual API editing received an excellent mean SUS score of 87.5 (Section 10.4).

Safe-DS library (Part II). The Safe-DS library is written in Python and offers a *simplified*, *integrated*, and *consistent* API for DS. The API is expressive enough to implement complete DS pipelines that involve tabular, time series, or image data. Data can be prepared (Section 16.3, Section 16.4), visualized (Section 16.5), and used to train supervised classical ML models (Section 16.7) and supervised NNs (Section 16.8). It forwards calls to established, efficient DS libraries, which greatly decreases development and maintenance effort.

The design of the API places a particular focus on *discoverability*, i.e. making it easy to find the API elements required to solve a given task, and *safety*, i.e. preventing mistakes entirely or at least detecting them early at runtime and helping users recover by offering useful error messages (Section 16.1). A high level of abstraction of the API provides a close mapping to the users' domain model, which is a cornerstone for discoverability.

The evaluation of the Safe-DS library (Section 18.2) revealed that compared to the original state-of-the-art DS libraries, it

- is more usable (plus 10 points in the mean SUS score),
- is easier to learn (plus 6 points in the mean SUS-like learnability score),
- lets DS novices get considerably more work done on their own (2 vs. 40 incomplete steps out of 119), and
- enables faster development (minus 22.6% mean required time with a generous imputation rule).

Safe-DS language (Part III). The Safe-DS language is a textual DSL for the development of DS pipelines. It combines the expressiveness of Python DS libraries with safety via compile-time error detection, without extra effort from the user.

Python DS libraries are integrated into the languages by a *stub language* (Chapter 24), which accurately captures their API using type annotations (Section 25.1), and constraints (Section 25.5). Constraints can express preconditions of functions that go beyond the type system, like *boundary constraints* for numbers (to prevent out-of-bounds values), *state constraints* for objects (to ensure that methods are called in the correct order), or *schema constraints* for tabular data (to prevent illegal column accesses). Stubs can be created largely automatically from a library's code and documentation (Section 24.4).

DS pipelines are written in a simple *pipeline language* (Chapter 23) as a sequence of assignments and calls of functions defined in stubs. All types needed for type checking (Section 25.4) get *inferred* from the pipeline code (Section 25.3). This also applies to all information needed for constraint checking.

The Safe-DS language has no dedicated evaluation, since that would require writing code in a plaintext editor or on a whiteboard, which is not a realistic scenario. Instead, it was evaluated together with the Safe-DS IDE (see below).

Safe-DS execution (Part IV). Our execution approach enables explorative development of DS pipelines without the pitfalls of notebooks, where users might forget to rerun code after changes or run it in the wrong order.

We derive a fine-grained dependency graph that focuses on individual operations rather than cells (Section 31.1). It also contains reads from and writes to files. Users then define up to which point they want to run their pipeline. From this, we derive an execution plan that is *correct* (contains all operations and runs them in a valid order) and *minimal* (does not contain unnecessary operations) by determining a backward slice (Section 31.2).

While running a pipeline, we store the results of calls to pure functions and functions that are only impure because they read from files in a memoization table. When a pipeline is rerun, potentially after making changes, we look up previously computed results to avoid expensive recomputation (Section 31.3). We also suggest various strategies for the deletion of entries from the memoization table, so it fits into restricted memory (Section 31.4).

The approach is used to efficiently run Safe-DS pipelines, without any further effort from the user: The stub language declares the purity of functions (Section 32.1), so it can be inferred in the pipeline language (Section 32.2). Safe-DS code is then translated to Python (Section 32.3) and executed by a runner (Section 32.4).

Safe-DS pipelines can be run almost as quickly as state-of-the-art notebook tools can run equivalent degraded Python notebooks that only contain one call per cell (Section 34.1). We only have a small constant overhead for generation of Python code and communication with the runner. The addition of memoization leads to a further drastic reduction in execution times when rerunning programs up to a factor of 212 (Section 34.2), given sufficient memory (Section 34.3).

Safe-DS IDE (Part V). Finally, we developed an IDE for the Safe-DS language. Language support for the textual pipeline and stub languages (Section 39.1) encompasses continuous static checking with helpful error messages, code-completion that filters suggestions based on the current context, tooltips to show documentation, inlay hints to show inferred information like types, and more.

All this functionality is available in a language server, so it can be integrated easily into modern editors and IDEs that support the Language Server Protocol. We also provide an extension for VS Code that bundles the language server. VS Code offers comprehensive language-agnostics features out-of-the-box, like a powerful text editor, and the integration of version control systems.

Beyond that, our VS Code extension includes a graphical view onto Safe-DS pipelines (Section 39.2). It can already display any syntactically valid pipeline. This cannot be done in a user-friendly manner for a general-purpose language like Python due to the amount of language concepts to cover. The comparatively simple addition of editing capabilities is future work. Then users will be able to switch between the text and graph view for any valid program, so they can always use the view best suited for their current task.

Pipelines can be partially executed by clicking on code lenses created by the language server, which triggers code generation (including the computation of a backward slice) and code execution with the runner (including memoization), as explained in Chapter 32. The computed results are shown in dedicated views in VS Code: Primitive values like strings are printed to a log, images are shown in a simple image viewer, and tables are opened in a custom table explorer.

The table explorer (Section 39.3) displays the data in the table, as well as statistics, quality metrics, and visualizations for all columns. The data can be sorted and filtered, and users can create additional plots in a simple dialog. Full integration of the table explorer into the same execution system we use for pipelines allows transparent reuse of results.

A usability study (Section 41.3) compared a typical development stack that consisted of Python, various DS libraries (Polars, Matplotlib, and scikit-learn), and a notebook environment to the full Safe-DS stack (library, language, execution concept, and IDE). It indicates that the full Safe-DS stack

- is considerably more usable (plus 16.4 points in the mean SUS score),
- is significantly easier to learn (plus 34.5 points in the mean SUS-like learnability score),
- lets DS novices get much more work done on their own (no vs. 40 incomplete steps out of 119), and
- greatly accelerates development (minus 26.3% mean required time with a generous imputation rule).

We, thus, conclude that Safe-DS is a valuable toolkit for the development of data science pipelines.

Bibliography

[1] V. Dhar, "Data science and prediction," *Commun. ACM*, vol. 56, no. 12, pp. 64–73, Dec. 2013, doi: 10.1145/2500499.

[2] D. Reinsel, J. Gantz, and J. Rydning, "The Digitization of the World from Edge to Core," 2018.

[3] S. Biswas, M. Wardat, and H. Rajan, "The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2091–2103, doi: 10.1145/3510003.3510057.

[4] L. Reimann and G. Kniesel-Wünsche, "Improving the Learnability of Machine Learning APIs by Semi-Automated API Wrapping," in *44th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results ICSE (NIER) 2022, Pittsburgh, PA, USA*, 2022, pp. 46–50, doi: 10.1109/ICSE-NIER55298.2022.9793507.

[5] L. Reimann and G. Kniesel-Wünsche, "Adaptoring: Adapter Generation to Provide an Alternative API for a Library," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024, Rovaniemi, Finland*, 2024, pp. 192–203, doi: 10.1109/SANER60148.2024.00027.

[6] C. R. B. de Souza and D. L. M. Bentolila, "Automatic Evaluation of API Usability Using Complexity Metrics and Visualizations," in *2009 31st International Conference on Software Engineering - Companion Volume*, May 2009, pp. 299–302, doi: 10.1109/ICSE-COMPANION.2009.5071006.

[7] W. Wang and M. W. Godfrey, "Detecting API usage obstacles: A study of iOS and Android developer questions," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, vol. 0, no. , pp. 61–64, doi: 10.1109/MSR.2013.6624006.

[8] M. Zibran, F. Eishita, and C. Roy, "Useful, But Usable? Factors Affecting the Usability of APIs," in *2011 18th Working Conference on Reverse Engineering*, 2011, pp. 151–155, doi: 10.1109/WCRE.2011.26.

[9] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703–732, 2011, doi: https://doi.org/10.1007/s10664-010-9150-8.

[10] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009, doi: 10.1109/MS.2009.193.

[11] M. Fowler, *Refactoring*, 2nd ed. Boston, MA: Addison-Wesley, 2018.

[12] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sep. 2005, pp. 389–398, doi: 10.1109/ICSM.2005.90.

[13] T. Mens and T. Tourwe, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, Feb. 2004, doi: 10.1109/TSE.2004.1265817.

[14] M. Abebe and C.-J. Yoo, "Trends, Opportunities and Challenges of Software Refactoring: A Systematic Literature Review," *International Journal of Software Engineering and Its Applications*, 2014.

[15] A. A. B. Baqais and M. Alshayeb, "Automatic Software Refactoring: A Systematic Literature Review," *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, Jun. 2020, doi: 10.1007/s11219-019-09477-y.

[16] J. Henkel and A. Diwan, "CatchUp! Capturing and Replaying Refactorings to Support API Evolution," in *Proceedings of the 27th International Conference on Software Engineering*, May 2005, pp. 274–283, doi: 10.1145/1062455.1062512.

[17] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *ECOOP 2006 – Object-Oriented Programming*, 2006, pp. 404–428, doi: 10.1007/11785477_24.

[18] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API Evolution under Control," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, Feb. 2014, pp. 420–424, doi: 10.1109/CSMR-WCRE.2014.6747209.

[19] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, Dec. 2007, doi: 10.1109/TSE.2007.70747.

[20] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A Graph-Based Approach to API Usage Adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2010, pp. 302–321, doi: 10.1145/1869459.1869486.

[21] J. H. Perkins, "Automatically Generating Refactorings to Support API Evolution," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Sep. 2005, pp. 111–114, doi: 10.1145/1108792.1108818.

[22] C. H. Kao, C.-Y. Chang, and H. C. Jiau, "Towards cost-effective API deprecation: A win–win strategy for API developers and API users," *Information and Software Technology*, vol. 142, p. 106746, 2022, doi: https://doi.org/10.1016/j.infsof.2021.106746.

[23] T. Scheller and E. Kühn, "Automated Measurement of API Usability," *Information and Software Technology*, vol. 61, no. C, pp. 145–162, May 2015, doi: 10.1016/j.infsof.2015.01.009.

[24] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[25] U. Jugel, "Generating Smart Wrapper Libraries for Arbitrary APIs," in *Software Language Engineering*, 2010, pp. 354–373, doi: 10.1007/978-3-642-12107-4_24.

[26] B. A. Myers and J. Stylos, "Improving API Usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, May 2016, doi: 10.1145/2896587.

[27] T. Scheller and E. Kuhn, "Influencing Factors on the Usability of API Classes and Methods," in *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, Apr. 2012, pp. 232–241, doi: 10.1109/ECBS.2012.27.

[28] "Typing — Support for Type Hints." https://docs.python.org/3/library/typing.html (accessed Sep. 19, 2024).

[29] H. Wright, "Hyrum's Law." https://www.hyrumslaw.com/ (accessed Jul. 15, 2023).

[30] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Aug. 2015, pp. 307–319, doi: 10.1145/2786805.2786852.

[31] T. Gowers, J. Barrow-Green, and I. Leader, Editors, *The Princeton Companion to Mathematics*. Princeton: Princeton University Press, 2008.

[32] E. W. Weisstein, "P-Value." https://mathworld.wolfram.com/ (accessed Sep. 19, 2024).

[33] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[34] L. Buitinck *et al.*, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[35] R. Nystrom, *Crafting Interpreters*. Genever Benning, 2021.

[36] "Word Classes and Phrase Classes," Aug. 28, 2024. https://dictionary.cambridge.org/grammar/british-grammar/word-classes-and-phrase-classes (accessed Sep. 02, 2024).

[37] "Clauses and Sentences," Aug. 28, 2024. https://dictionary.cambridge.org/grammar/british-grammar/clauses-and-sentences (accessed Sep. 02, 2024).

[38] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python," 2020, doi: 10.5281/zenodo.1212303.

[39] "Style Guide — Numpydoc v1.6.0rc2.Dev0 Manual." https://numpydoc.readthedocs.io/en/latest/format.html (accessed Jul. 26, 2023).

[40] "Styleguide." https://google.github.io/styleguide/pyguide.html (accessed Jul. 26, 2023).

[41] "reStructuredText Primer — Sphinx Documentation." https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html (accessed Jul. 26, 2023).

[42] "The Epytext Markup Language." https://epydoc.sourceforge.net/epytext.html (accessed Jul. 26, 2023).

[43] "8. Compound Statements." https://docs.python.org/3/reference/compound_stmts.html (accessed Sep. 04, 2024).

[44] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering - ASE '05*, 2005, p. 54, doi: 10.1145/1101908.1101919.

[45] Z. Xing and E. Stroulia, "API-Evolution Support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, Dec. 2007, doi: 10.1109/TSE.2007.70747.

[46] B. Dagenais and M. P. Robillard, "Recommending Adaptive Changes for Framework Evolution," in *Proceedings of the 30th International Conference on Software Engineering*, May 2008, pp. 481–490, doi: 10.1145/1368088.1368154.

[47] B. Dagenais and M. P. Robillard, "SemDiff: Analysis and Recommendation Support for API Evolution," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 599–602, doi: 10.1109/ICSE.2009.5070565.

[48] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A Hybrid Approach to Identify Framework Evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, May 2010, pp. 325–334, doi: 10.1145/1806799.1806848.

[49] S. Meng, X. Wang, L. Zhang, and H. Mei, "A History-Based Matching Approach to Identification of Framework Evolution," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 353–363, doi: 10.1109/ICSE.2012.6227179.

[50] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, 2002, doi: 10.1109/TSE.2002.1000449.

[51] N. Vollroth, “Extraktion von Vorbedingungen aus Python-Docstrings,” Bonn, Germany, 2023.

[52] S. Milton and H. W. Schmidt, “Dynamic Dispatch in Object-Oriented Languages,” 1994.

[53] “Importlib — The Implementation of Import.” https://docs.python.org/3/library/importlib.html (accessed Jul. 15, 2023).

[54] Y. Futamura, “Partial Computation of Programs,” in *RIMS Symposia on Software Science and Engineering*, 1983, pp. 1–35, doi: 10.1007/3-540-11980-9_13.

[55] X. Wang and L. Zhao, “APICAD: Augmenting API Misuse Detection through Specifications from Code and Documents,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, vol. 0, no. , pp. 245–256, doi: 10.1109/ICSE48619.2023.00032.

[56] “Indexed Database API 3.0.” https://www.w3.org/TR/IndexedDB/ (accessed Sep. 19, 2024).

[57] “IndexedDB API - Web APIs | MDN,” Aug. 08, 2024. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (accessed Sep. 19, 2024).

[58] “HTML Standard.” https://html.spec.whatwg.org/multipage/nav-history-apis.html#the-history-interface (accessed Sep. 19, 2024).

[59] “History API - Web APIs | MDN,” Jul. 26, 2024. https://developer.mozilla.org/en-US/docs/Web/API/History_API (accessed Sep. 19, 2024).

[60] “Java Virtual Machine Technology Overview.” https://docs.oracle.com/en/java/javase/22/vm/java-virtual-machine-technology-overview.html (accessed Sep. 19, 2024).

[61] M. Akhin and M. Belyaev, “Kotlin Language Specification.” https://kotlinlang.org/spec/introduction.html (accessed Sep. 19, 2024).

[62] “PyCharm: The Python IDE for Data Science and Web Development.” https://www.jetbrains.com/pycharm/ (accessed Dec. 14, 2024).

[63] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA, US: The MIT Press, 1984, p. 426.

[64] T. Boren and J. Ramey, “Thinking Aloud: Reconciling Theory and Practice,” *IEEE Transactions on Professional Communication*, vol. 43, no. 3, pp. 261–278, Sep. 2000, doi: 10.1109/47.867942.

[65] E. Charters, “The Use of Think-aloud Methods in Qualitative Research An Introduction to Think-aloud Methods,” *Brock Education Journal*, vol. 12, no. 2, Jul. 2003, doi: 10.26522/brocked.v12i2.38.

[66] M. W. M. Jaspers, T. Steen, C. van den Bos, and M. Geenen, “The Think Aloud Method: A Guide to User Interface Design,” *International Journal of Medical Informatics*, vol. 73, no. 11, pp. 781–795, Nov. 2004, doi: 10.1016/j.ijmedinf.2004.08.003.

[67] K. A. Young, “Direct from the Source: The Value of 'think-Aloud' Data in Understanding Learning,” *Journal of Educational Enquiry*, vol. 6, p. 10, 2009.

[68] D. W. Eccles and G. Arsal, “The Think Aloud Method: What Is It and How Do I Use It?,” *Qualitative Research in Sport, Exercise and Health*, vol. 9, no. 4, pp. 514–531, Aug. 2017, doi: 10.1080/2159676X.2017.1331501.

[69] J. Brooke, “SUS - A Quick and Dirty Usability Scale,” *Usability Eval. Ind.*, vol. 189, p. 7, 1995.

[70] J. R. Lewis, “The System Usability Scale: Past, Present, and Future,” *International Journal of Human–Computer Interaction*, vol. 34, no. 7, pp. 577–590, Jul. 2018, doi: 10.1080/10447318.2018.1455307.

[71] E. W. Weisstein, “Bessel's Correction.” https://mathworld.wolfram.com/BesselsCorrection.html (accessed Jan. 13, 2025).

[72] A. Bangor, P. T. Kortum, and J. T. Miller, “An Empirical Evaluation of the System Usability Scale,” *International Journal of Human–Computer Interaction*, vol. 24, no. 6, pp. 574–594, Jul. 2008, doi: 10.1080/10447310802205776.

[73] A. Bangor, P. Kortum, and J. Miller, “Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale,” *J. Usability Studies*, vol. 4, no. 3, pp. 114–123, May 2009.

[74] S. Mclellan, A. Muddimer, and S. Peres, “The Effect of Experience on System Usability Scale Ratings,” *Journal of Usability Studies*, vol. 7, Nov. 2011.

[75] “PEP 570 – Python Positional-Only Parameters | Peps.Python.Org.” https://peps.python.org/pep-0570/ (accessed Sep. 23, 2024).

[76] “PEP 3102 – Keyword-Only Arguments | Peps.Python.Org.” https://peps.python.org/pep-3102/ (accessed Sep. 23, 2024).

[77] T. Grill, O. Polacek, and M. Tscheligi, “Methods towards API Usability: A Structural Analysis of Usability Problem Categories,” in *Human-Centered Software Engineering*, 2012, pp. 164–180.

[78] R. Lämmel, E. Pek, and J. Starek, “Large-Scale, AST-based API-usage Analysis of Open-Source Java Projects,” in *Proceedings of the 2011 ACM Symposium on Applied Computing*, Mar. 2011, pp. 1317–1324, doi: 10.1145/1982185.1982471.

[79] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Prentice Hall, 2009.

[80] S. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[81] J. Stylos and S. Clarke, “Usability Implications of Requiring Parameters in Objects' Constructors,” in *Proceedings of the 29th International Conference on Software Engineering*, May 2007, pp. 529–539, doi: 10.1109/ICSE.2007.92.

[82] C. R. B. de Souza and D. L. M. Bentolila, “Automatic Evaluation of API Usability Using Complexity Metrics and Visualizations,” in *2009 31st International Conference on Software Engineering - Companion Volume*, May 2009, pp. 299–302, doi: 10.1109/ICSE-COMPANION.2009.5071006.

[83] G. M. Rama and A. Kak, “Some Structural Measures of API Usability: SOME STRUCTURAL MEASURES OF API USABILITY,” *Software: Practice and Experience*, vol. 45, no. 1, pp. 75–110, Jan. 2015, doi: 10.1002/spe.2215.

[84] Q. Xin, M. Kim, Q. Zhang, and A. Orso, “Subdomain-Based Generality-Aware Debloating,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, vol. 0, no. , pp. 224–236.

[85] Q. Xin, Q. Zhang, and A. Orso, “Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating,” 2023, doi: 10.1145/3551349.3556970.

[86] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, “An Empirical Study on Configuration Errors in Commercial and Open Source Systems,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Oct. 2011, pp. 159–172, doi: 10.1145/2043556.2043572.

[87] V. Ramachandran, M. Gupta, M. Sethi, and S. R. Chowdhury, “Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-Tiered Enterprise Applications,” in *Proceedings of the 6th International Conference on Autonomic Computing*, Jun. 2009, pp. 169–178, doi: 10.1145/1555228.1555269.

[88] X. Liao, S. Zhou, S. Li, Z. Jia, X. Liu, and H. He, "Do You Really Know How to Configure Your Software? Configuration Constraints in Source Code May Help," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 832–846, Sep. 2018, doi: 10.1109/TR.2018.2834419.

[89] S. Zhang and M. D. Ernst, "Which Configuration Option Should I Change?," in *Proceedings of the 36th International Conference on Software Engineering*, May 2014, pp. 152–163, doi: 10.1145/2568225.2568251.

[90] J. Bloch, "How to Design a Good API and Why It Matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, Oct. 2006, pp. 506–507, doi: 10.1145/1176617.1176622.

[91] D. Hou and L. Li, "Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions," in *2011 IEEE 19th International Conference on Program Comprehension*, Jun. 2011, pp. 91–100, doi: 10.1109/ICPC.2011.21.

[92] T. Grill, O. Polacek, and M. Tscheligi, "Methods towards API Usability: A Structural Analysis of Usability Problem Categories," in *Human-Centered Software Engineering*, 2012, pp. 164–180.

[93] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "API Designers in the Field: Design Practices and Challenges for Creating Usable APIs," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2018, pp. 249–258, doi: 10.1109/VLHCC.2018.8506523.

[94] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations," *Journal of Systems and Software*, vol. 167, p. 110610, Sep. 2020, doi: 10.1016/j.jss.2020.110610.

[95] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A Systematic Review on the Code Smell Effect," *Journal of Systems and Software*, vol. 144, pp. 450–477, Oct. 2018, doi: 10.1016/j.jss.2018.07.035.

[96] A. Yamashita and L. Moonen, "Do Developers Care about Code Smells? An Exploratory Survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct. 2013, pp. 242–251, doi: 10.1109/WCRE.2013.6671299.

[97] F. Arcelli Fontana, P. Braione, and M. Zanoni, "Automatic Detection of Bad Smells in Code: An Experimental Assessment," *Journal of Object Technology*, vol. 11, Jan. 2012, doi: 10.5381/jot.2012.11.2.a5.

[98] J. Al Dallal, "Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review," *Information and Software Technology*, vol. 58, pp. 231–249, Feb. 2015, doi: 10.1016/j.infsof.2014.08.002.

[99] G. Rasool and Z. Arshad, "A Review of Code Smell Mining Techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015, doi: 10.1002/smr.1737.

[100] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting Code Smells Using Machine Learning Techniques: Are We There Yet?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2018, pp. 612–621, doi: 10.1109/SANER.2018.8330266.

[101] L. Reimann and G. Kniesel-Wünsche, "Achieving Guidance in Applied Machine Learning through Software Engineering Techniques," in *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal*, 2020, pp. 7–12, doi: 10.1145/3397537.3397552.

[102] Kaggle, "2022 Kaggle Machine Learning & Data Science Survey." https://kaggle.com/competitions/kaggle-survey-2022 (accessed Oct. 25, 2024).

[103] The pandas development team, "pandas-dev/pandas: Pandas." https://github.com/pandas-dev/pandas.

[104] D. Sculley *et al.*, "Hidden technical debt in Machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, 2015, pp. 2503–2511.

[105] A. Jovic, K. Brkic, and N. Bogunovic, "An Overview of Free Software Tools for General Data Mining," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2014, pp. 1112–1117, doi: 10.1109/MIPRO.2014.6859735.

[106] G. Nguyen *et al.*, "Machine Learning and Deep Learning Frameworks and Libraries for Large-Scale Data Mining: A Survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, Jun. 2019, doi: 10.1007/s10462-018-09679-z.

[107] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, doi: 10.1109/MCSE.2007.55.

[108] Polars maintainers and contributors, "Polars: Blazingly fast DataFrames in Rust, Python, Node.js, R, and SQL," 2021. https://github.com/pola-rs/polars.

[109] J. VanderPlas *et al.*, "Altair: Interactive Statistical Visualizations for Python," *Journal of Open Source Software*, vol. 3, no. 32, p. 1057, 2018, doi: 10.21105/joss.01057.

[110] Pillow maintainers, "Pillow - Python Imaging Library (Fork)," 2015. https://github.com/python-pillow/Pillow.

[111] Pillow-SIMD maintainers, "Pillow-SIMD - The friendly PIL fork," 2010. https://github.com/uploadcare/pillow-simd.

[112] S. van der Walt *et al.*, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 2014, doi: 10.7717/peerj.453.

[113] C. R. Harris *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, doi: 10.1038/s41586-020-2649-2.

[114] TorchVision maintainers and contributors, "TorchVision: PyTorch's Computer Vision library," 2016. https://github.com/pytorch/vision.

[115] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., pp. 8024–8035, 2019, [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[116] C. Shorten and T. M. Khoshgoftaar, "A Survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, no. 1, p. 60, Jul. 2019, doi: 10.1186/s40537-019-0197-0.

[117] Martín~Abadi and others, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. https://www.tensorflow.org/.

[118] Matplotlib development team, "Matplotlib Application Interfaces (APIs) — Matplotlib 3.9.2 Documentation." https://matplotlib.org/stable/users/explain/figure/api_interfaces.html#why-be-explicit (accessed Nov. 13, 2024).

[119] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021, doi: 10.21105/joss.03021.

[120] Plotly Technologies Inc., "Plotly Python Graphing Library," 2015. https://plotly.com/python/.

[121] Bokeh Development Team, "Bokeh: Python library for interactive visualization." 2018, [Online]. Available: https://bokeh.pydata.org/en/latest/.

[122] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[123] scikit-learn development team, "scikit-learn FAQ." https://scikit-learn.org/stable/faq.html#will-you-add-gpu-support (accessed Nov. 13, 2024).

[124] J. H. Friedman, "Greedy function approximation: A gradient boosting machine.," *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001, doi: 10.1214/aos/1013203451.

[125] J. H. Friedman, "Stochastic gradient boosting," *Comput. Stat. Data Anal.*, vol. 38, no. 4, pp. 367–378, Feb. 2002, doi: 10.1016/S0167-9473(01)00065-2.

[126] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794, doi: 10.1145/2939672.2939785.

[127] G. Ke *et al.*, "LightGBM: a highly efficient gradient boosting decision tree," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 3149–3157.

[128] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: unbiased boosting with categorical features," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 6639–6649.

[129] J. Bradbury *et al.*, "JAX: composable transformations of Python+NumPy programs," 2018. http://github.com/jax-ml/jax.

[130] J. Heek *et al.*, "Flax: A neural network library and ecosystem for JAX," 2024. http://github.com/google/flax.

[131] F. Chollet and others, "Keras," 2015. https://keras.io/.

[132] T. Wolf *et al.*, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Oct. 2020, pp. 38–45, doi: 10.18653/v1/2020.emnlp-demos.6.

[133] mypy Development Team, "Mypy: Static Typing for Python," 2014. https://github.com/python/mypy.

[134] "3. Data Model." https://docs.python.org/3/reference/datamodel.html (accessed Nov. 18, 2024).

[135] "4. More Control Flow Tools." https://docs.python.org/3/tutorial/controlflow.html (accessed Nov. 18, 2024).

[136] "PEP 692 – Using TypedDict for More Precise **kwargs Typing | Peps.Python.Org." https://peps.python.org/pep-0692/ (accessed Nov. 18, 2024).

[137] S. Breuer, "Benutzerfreundliche Feed-Forward Neural Networks für das Lernen von Tabellendaten," Bonn, Germany, 2024.

[138] A. Gerhards, "Implementierung einer nutzerfreundlichen Schnittstelle für das Vorhersagen von Zeitreihen mittels rekurrenter neuronaler Netze," Bonn, Germany, 2024.

[139] A. F. Gréus, "Benutzerfreundliche Convolutional Neural Networks für das Lernen auf Bilddaten," Bonn, Germany, 2024.

[140] C. Grefrath, "Evaluation der Safe-DS Bibliothek und Sprache (Working Title)," Bonn, Germany, 2025.

[141] "Visual Studio Code - Code Editing. Redefined." https://code.visualstudio.com/ (accessed Dec. 14, 2024).

[142] J. R. Lewis and J. Sauro, "The Factor Structure of the System Usability Scale," in *Human Centered Design*, 2009, pp. 94–103.

[143] L. Reimann and G. Kniesel-Wünsche, "Safe-DS: A Domain Specific Language to Make Data Science Safe," in *45th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE, Melbourne, Australia*, 2023, pp. 72–77, doi: 10.1109/ICSE-NIER58687.2023.00019.

[144] B. van Oort, L. Cruz, M. Aniche, and A. van Deursen, " The Prevalence of Code Smells in Machine Learning projects ," in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, May 2021, vol. 0, pp. 1–8, doi: 10.1109/WAIN52551.2021.00011.

[145] "Coroutines and Tasks." https://docs.python.org/3/library/asyncio-task.html (accessed Nov. 25, 2024).

[146] Wikipedia contributors, "Wizard (software) — Wikipedia, The Free Encyclopedia," 2024. https://en.wikipedia.org/w/index.php?title=Wizard_(software)&oldid=1258609875 (accessed Nov. 28, 2024).

[147] RapidMiner, "RapidMiner Go." https://docs.rapidminer.com/9.9/go/overview/index.html (accessed Nov. 28, 2024).

[148] Kotlin DataFrame development team, "Kotlin DataFrame: typesafe in-memory structured data processing for JVM," 2023. https://github.com/Kotlin/dataframe.

[149] T. Kluyver *et al.*, "Jupyter Notebooks - a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, pp. 87–90, [Online]. Available: https://eprints.soton.ac.uk/403913/.

[150] A. Solbach, "Developing a Domain Specific Language for the Evolving Field of Machine Learning," Weimar, Germany, 2024.

[151] M. Voelter *et al.*, *DSL Engineering — Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[152] F. Campagne, *The MPS Language Workbench: Volume I*, Third Edition. Fabien Campagne, 2016.

[153] F. Campagne, *The MPS Language Workbench: Volume II*, First Edition. Fabien Campagne, 2016.

[154] F. Campagne, W. E. Digan, and M. Simi, "MetaR: simple, high-level languages for data analysis with the R ecosystem," *bioRxiv*, 2016, doi: 10.1101/030254.

[155] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "YALE: Rapid Prototyping for Complex Data Mining Tasks," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '06*, 2006, p. 935, doi: 10.1145/1150402.1150531.

[156] "Altair Announces Completion of Acquisition of RapidMiner." https://altair.com/newsroom/news-releases/altair-announces-completion-of-acquisition-of-rapidminer (accessed Nov. 28, 2024).

[157] L. Reimann, “Safe-DS DSL - Operations.” https://dsl.safeds.com/en/stable/pipeline-language/expressions/operations/ (accessed Dec. 25, 2024).

[158] G. J. Sussman and G. L. Steele, “SCHEME: An Interpreter for Extended Lambda Calculus,” Cambridge, UK, Dec. 1975.

[159] H. G. Rice, “Classes of Recursively Enumerable Sets and Their Decision Problems,” *Transactions of the American Mathematical Society*, vol. 74, pp. 358–366, 1953.

[160] J. Alama and J. Korbmacher, “The Lambda Calculus,” *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2023.

[161] S. Klabnik and C. Nichols, “Defining an Enum - The Rust Programming Language.” https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html.

[162] “Built-in Functions.” https://docs.python.org/3/library/functions.html (accessed Nov. 30, 2024).

[163] “JavaDoc Documentation Comment Specification for the Standard Doclet (JDK 23).” https://docs.oracle.com/en/java/javase/23/docs/specs/javadoc/doc-comment-spec.html (accessed Nov. 30, 2024).

[164] Wikipedia contributors, “Markdown — Wikipedia, The Free Encyclopedia,” 2024. https://en.wikipedia.org/w/index.php?title=Markdown&oldid=1260346563 (accessed Nov. 30, 2024).

[165] “Built-in Package Support in Python 1.5.” https://www.python.org/doc/essays/packages/ (accessed Nov. 30, 2024).

[166] T. Freeman and F. Pfenning, “Refinement types for ML,” in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991, pp. 268–277, doi: 10.1145/113445.113468.

[167] M. Akhin and M. Belyaev, “Type System - Kotlin Language Specification.” https://kotlinlang.org/spec/type-system.html (accessed Dec. 01, 2024).

[168] “Kinds of Types - Mypy 1.13.0 Documentation.” https://mypy.readthedocs.io/en/stable/kinds_of_types.html (accessed Dec. 02, 2024).

[169] L. Cardelli, “Type Systems,” *CRC Handbook of Computer Science and Engineering*. CRC Press, 2004.

[170] International Organization for Standardization, Editor, “ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF.” 1996.

[171] G. D. Plotkin, “A Structural Approach to Operational Semantics,” University of Aarhus, 1981. [Online]. Available: http://citeseer.ist.psu.edu/plotkin81structural.html.

[172] M. Abadi and L. Cardelli, *A Theory of Objects*. New York, NY: Springer, 1996.

[173] S.-A. Islami, “Typextraktion aus Type Hints & DocStrings von Python Bibliotheken (Working Title),” Bonn, Germany, 2025.

[174] Langium development team, “Langium: Next-gen language engineering framework,” 2022. https://github.com/eclipse-langium/langium.

[175] Griffe Developers Team, “Griffe,” 2023. https://github.com/mkdocstrings/griffe.

[176] Pyright Development Team, “Pyright: Static Typing for Python,” 2019. https://github.com/microsoft/pyright.

[177] Pyre Development Team, “Pyre,” 2019. https://github.com/facebook/pyre-check.

[178] N. Bantilan *et al.*, "unionai-oss/pandera: Beta release v0.12.0b0," Aug. 2022. https://doi.org/10.5281/zenodo.6985652.

[179] B. van Oort, L. Cruz, B. Loni, and A. van Deursen, "```Project Smells'' - Experiences in Analysing the Software Quality of ML Projects with Mllint," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2022, pp. 211–220, doi: 10.1109/ICSE-SEIP55303.2022.9794115.

[180] P. Molino, Y. Dudin, and S. S. Miryala, "Ludwig: a type-based declarative deep learning toolbox," 2019. https://arxiv.org/abs/1909.07930.

[181] J. Giner-Miguelez, A. Gómez, and J. Cabot, "A domain-specific language for describing machine learning datasets," *Journal of Computer Languages*, vol. 76, p. 101209, 2023, doi: https://doi.org/10.1016/j.cola.2023.101209.

[182] B. Jahić, N. Guelfi, and B. Ries, "SEMKIS-DSL: A Domain-Specific Language to Support Requirements Engineering of Datasets and Neural Network Recognition," *Information*, vol. 14, no. 4, 2023, doi: 10.3390/info14040213.

[183] A. de la Vega, D. García-Saiz, M. Zorrilla, and P. Sánchez, "Lavoisier: A DSL for increasing the level of abstraction of data selection and formatting in data mining," *Journal of Computer Languages*, vol. 60, p. 100987, 2020, doi: https://doi.org/10.1016/j.cola.2020.100987.

[184] I. Portugal, P. Alencar, and D. Cowan, "A Survey on Domain-Specific Languages for Machine Learning in Big Data," *arXiv:1602.07637 [cs]*, Mar. 2016.

[185] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-so-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Jun. 2008, pp. 1099–1110, doi: 10.1145/1376616.1376726.

[186] "MapReduce Tutorial." https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html (accessed Dec. 04, 2024).

[187] N. G. Nanvou Tsopgny, T. Messi Nguélé, and E. Kouakam, "DSL for parallelizing Machine Learning algorithms on multicore architecture," Jul. 2022, [Online]. Available: https://hal.science/hal-03727658.

[188] T. Zhao and X. Huang, "Design and implementation of DeepDSL: A DSL for deep learning," *Computer Languages, Systems & Structures*, vol. 54, pp. 39–70, 2018, doi: https://doi.org/10.1016/j.cl.2018.04.004.

[189] A. Agrawal *et al.*, "TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning," in *Proceedings of Machine Learning and Systems*, 2019, vol. 1, pp. 178–189, [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2019/file/b3cd73d353d39e5cf6f6e9ff8d14c87f-Paper.pdf.

[190] A. Podobas *et al.*, "StreamBrain: An HPC Framework for Brain-like Neural Networks on CPUs, GPUs and FPGAs," 2021, doi: 10.1145/3468044.3468052.

[191] F. J. M. González, R. Rodriguez-Echeverria, J. M. Conejero, A. E. Prieto, and J. D. G. Rodriguez-Echeverriaz, "A Model-Driven Approach for Systematic Reproducibility and Replicability of Data Science Projects," in *Advanced Information Systems Engineering - 34th International Conference, CAiSE 2022, Leuven, Belgium, June 6-10, 2022, Proceedings*, 2022, vol. 13295, pp. 147–163, doi: 10.1007/978-3-031-07472-1_9.

[192] J. Zucker and M. d'Leeuwen, "Arbiter: A Domain-Specific Language for Ethical Machine Learning," in *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, 2020, pp. 421–425, doi: 10.1145/3375627.3375858.

[193] L. Reimann and G. Kniesel-Wünsche, "An Alternative to Cells for Selective Execution of Data Science Pipelines," in *45th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE, Melbourne, Australia*, 2023, pp. 129–134, doi: 10.1109/ICSE-NIER58687.2023.00029.

[194] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, Jan. 1984, doi: 10.1093/comjnl/27.2.97.

[195] T. Kluyver *et al.*, "Jupyter Notebooks — a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016, pp. 87–90, [Online]. Available: https://eprints.soton.ac.uk/403913/.

[196] Google, "Google Colaboratory." https://colab.research.google.com/ (accessed Oct. 02, 2022).

[197] Kaggle, "Notebooks Documentation." https://www.kaggle.com/docs/notebooks (accessed Oct. 02, 2022).

[198] Amazon, "Use Amazon SageMaker Notebook Instances - Amazon SageMaker." https://docs.aws.amazon.com/sagemaker/latest/dg/nbi.html (accessed Oct. 02, 2022).

[199] Jetbrains, "Jupyter Notebook Support | PyCharm." https://www.jetbrains.com/help/pycharm/jupyter-notebook-support.html (accessed Oct. 02, 2022).

[200] T. Kabir, "Notebooks, Visual Studio Code Style." https://code.visualstudio.com/blogs/2021/11/08/custom-notebooks (accessed Oct. 02, 2022).

[201] B. E. Granger, J. Grout, and B. Lp, "JupyterLab: Building Blocks for Interactive Computing." http://archive.ipython.org/media/SciPy2016JupyterLab.pdf (accessed Oct. 02, 2022).

[202] J. Grus, "I Don't Like Notebooks." https://docs.google.com/presentation/d/1n2RlMdmv1p25Xy5thJUhkKGvjtV-dkAIsUXP-AL4ffI (accessed Sep. 28, 2022).

[203] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, Apr. 2018, pp. 1–11, doi: 10.1145/3173574.3173748.

[204] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing Messes in Computational Notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, May 2019, pp. 1–12, doi: 10.1145/3290605.3300500.

[205] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 507–517, doi: 10.1109/MSR.2019.00077.

[206] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, Apr. 2020, pp. 1–12, doi: 10.1145/3313831.3376729.

[207] S. Macke, H. Gong, D. J.-L. Lee, A. Head, D. Xin, and A. Parameswaran, "Fine-Grained Lineage for Safer Notebook Interactions," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1093–1101, Feb. 2021, doi: 10.14778/3447689.3447712.

[208] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks," *Empirical Software Engineering*, vol. 26, no. 4, p. 65, May 2021, doi: 10.1007/s10664-021-09961-9.

[209] D. Michie, "Memo" Functions and Machine Learning," *Nature*, vol. 218, no. 5136, pp. 19–22, Apr. 1968, doi: 10.1038/218019a0.

[210] A. Agrawal and M. Scolnick, "marimo - an open-source reactive notebook for Python," Aug. 2023. https://github.com/marimo-team/marimo.

[211] D. Koop and J. Patel, "Dataflow Notebooks: Encoding and Tracking Dependencies of Cells," Jun. 2017, [Online]. Available: https://www.usenix.org/conference/tapp17/workshop-program/presentation/koop.

[212] "Nodebook | Stitch Fix Technology – Multithreaded." https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/ (accessed Oct. 05, 2022).

[213] S. Titov, Y. Golubev, and T. Bryksin, "ReSplit: Improving the Structure of Jupyter Notebooks by Re-Splitting Their Cells," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2022, pp. 492–496, doi: 10.1109/SANER53432.2022.00066.

[214] "What Is an Image?." https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/ (accessed Dec. 06, 2024).

[215] Wikipedia contributors, "Copy-on-write — Wikipedia, The Free Encyclopedia," 2024. https://en.wikipedia.org/w/index.php?title=Copy-on-write&oldid=1258225312 (accessed Dec. 07, 2024).

[216] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, no. 4, pp. 352–357, 1984, doi: 10.1109/TSE.1984.5010248.

[217] "Haskell Language." https://www.haskell.org/ (accessed Dec. 07, 2024).

[218] HaskellWiki, "Memoization — HaskellWiki, ," 2014. https://wiki.haskell.org/index.php?title=Memoization&oldid=57978 (accessed Dec. 07, 2024).

[219] Wikipedia contributors, "Fibonacci sequence — Wikipedia, The Free Encyclopedia," 2024. https://en.wikipedia.org/w/index.php?title=Fibonacci_sequence&oldid=1261560902 (accessed Dec. 07, 2024).

[220] J. Yeen, "What Are Source Maps?." https://web.dev/articles/source-maps (accessed Dec. 09, 2024).

[221] L. Radermacher, "Statische Purity-Analyse für Python-Funktionen," Bonn, Germany, 2024.

[222] W. Oberländer, "Ausführung von Data-Science-Programmen," Bonn, Germany, 2024.

[223] WHATWG, "WebSockets Standard," Sep. 25, 2024. https://websockets.spec.whatwg.org/ (accessed Dec. 09, 2024).

[224] "Multiprocessing.Shared_memory — Shared Memory for Direct Access across Processes." https://docs.python.org/3/library/multiprocessing.shared_memory.html (accessed Dec. 09, 2024).

[225] K. R. Davis, B. Peabody, and P. Leach, "Universally Unique IDentifiers (UUIDs)," May 2024. doi: 10.17487/RFC9562.

[226] Gradle development team, "Gradle - Adaptable, fast automation for all," 2008. https://github.com/gradle/gradle.

[227] Apache Airflow development team, "Apache Airflow - A platform to programmatically author, schedule, and monitor workflows," 2015. https://github.com/apache/airflow.

[228] D. Coss, "The CIA strikes back: Redefining confidentiality, integrity and availability in security," *Journal of Information System Security*, vol. 10, p. , 2014.

[229] LSP Contributors, "Official Page for Language Server Protocol." https://microsoft.github.io/language-server-protocol/ (accessed Sep. 08, 2022).

[230] "Amazon SageMaker Canvas." https://aws.amazon.com/sagemaker-ai/canvas/ (accessed Dec. 14, 2024).

[231] R. Du *et al.*, "Rapsai: Accelerating Machine Learning Prototyping of Multimedia Applications through Visual Programming," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, Apr. 2023, pp. 1–23, doi: 10.1145/3544548.3581338.

[232] "Dataiku." https://www.dataiku.com/ (accessed Dec. 14, 2024).

[233] "KNIME." https://www.knime.com/ (accessed Dec. 14, 2024).

[234] J. Demšar *et al.*, "Orange: Data Mining Toolbox in Python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013, [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html.

[235] "Integrating Your Objects with IPython — IPython 8.30.0 Documentation." https://ipython.readthedocs.io/en/stable/config/integrating.html (accessed Dec. 14, 2024).

[236] "Datalore – Collaborative Data Science Platform by JetBrains." https://www.jetbrains.com/datalore/ (accessed Dec. 14, 2024).

[237] Data Wrangler development team, "Data Wrangler Extension for Visual Studio Code," 2023. https://github.com/microsoft/vscode-data-wrangler.

[238] SandDance development team, "SandDance - Visually explore, understand, and present your data. ," 2019. https://github.com/Microsoft/Sanddance.

[239] "Business Intelligence and Analytics Software | Tableau." https://www.tableau.com/ (accessed Dec. 15, 2024).

[240] "Power BI - Data Visualization | Microsoft Power Platform." https://www.microsoft.com/en-us/power-platform/products/power-bi (accessed Dec. 15, 2024).

[241] LSP Contributors, "Language Server Protocol Specification - 3.18." https://microsoft.github.io/language-server-protocol/specifications/lsp/3.18/specification/ (accessed Dec. 19, 2024).

[242] B. Bothin, "Integrated Exploratory Data Analysis for Development Environments," Bonn, Germany, 2024.

[243] G. König, "Graphische Repräsentation für Safe-DS Pipelines (Working Title)," Bonn, Germany, 2025.

[244] "Webview API." https://code.visualstudio.com/api/extension-guides/webview (accessed Dec. 20, 2024).

Figures

Figure 1: The DS process is mostly linear but has built-in feedback loops [3]. 1

Figure 2: The complete Safe-DS stack. Adaptoring helped create the Safe-DS library. 2

Figure 3: The discussed approaches to create a new API for a library. Blue boxes stand for the original library and green boxes for the new one. Original users (👶) interact with the original function until they migrate their code. New users (👶) directly use the changed function. Adaptoring builds the new API based on the unmodified, original one. 5

Figure 4: UML class diagram of a variant of the adapter pattern that uses object composition [24]. The client wants to call `Target.changed`, but only `Adaptee.original` exists. The adapter bridges this gap. 7

Figure 5: Part of speech tags and dependency tree for a parameter dependency (classified by spaCy). The action is marked in red, the condition in blue. Action and condition are linked by a dependency of type adverbial clause modifier (purple). 13

Figure 6: An excerpt of the GUI for review and extension of API transformations. Users can navigate the API elements using the *tree view* on the left and then inspect information about them in the selection view on the right. 17

Figure 7: Dialog to add or edit a transformation that sets the default value of a parameter. 17

Figure 8: Buttons for transformation review. API elements can be marked as complete, indicating that all transformations are specified. Each transformation can be marked as correct, unsure, or wrong. 19

Figure 9: Visualization of the number of remaining parameters for sensible usefulness thresholds. 19

Figure 10: Projects (gray) and components (black) that encompass the proof-of-concept implementation of the adaptoring approach for Python. 20

Figure 11: Number of useful, useless, and unused elements in the public API of scikit-learn. Classes and functions are useful if and only if they are used. 23

Figure 12: Number of public API elements of scikit-learn with higher usage count/usefulness. 24

Figure 13: Comparison of the System Usability Scores for the API Editor GUI vs. PyCharm with GitHub Copilot for the creation of adapters. 28

Figure 14: The task-oriented package structure of the Safe-DS API. 48

Figure 15: A simple example table, with columns (red), rows (blue), and cells (purple). 49

Figure 16: Finding edges in an image via a method of the `Image` class. 52

Figure 17: A histogram created for a single `Column`. 53

Figure 18: Histograms created for all columns of a `Table` at once and arranged in a grid. 53

Figure 19: Constructing a `TabularDataset` from a `Table`. ML models predict the `target` column based on the `features`. The `extra` columns are ignored by models during training. During inference they are included unchanged in the output to associate predictions with identifiers. 54

Figure 20: An `ImageDataset` that contains an `ImageList` as features and a `Column` as target. 54

Figure 21: UML class diagram for the hierarchy of the classical supervised models in the Safe-DS API. For brevity, not all metrics methods are shown, and concrete classes, which inherit from `Classifier` or `Regressor`, are omitted. 55

Figure 22: UML class diagram of the composite pattern [24]. The `Client` can treat `Leaf` and `Composite` objects the same and only interacts with them via the `Component` interface. 59

Figure 23: The implementation of the abstract `ImageList` class using a derivation of the composite design pattern shown in Figure 22. Major changes are marked in blue. The `add` and `remove` methods return a new `ImageList` object, which allows them to switch between concrete implementations. . 59

Figure 24: A box plot of the SUS scores for the used libraries. Higher values are better. 63

Figure 25: A box plot of the SUS-like scores about the learnability of the used libraries. It is based on two items of the normal SUS questionnaire, and a custom one. The final score is between 0 and 100. Higher values are better. 63

Figure 26: A box plot of the time (in minutes) to solve the task with original libraries (Group 1) and the Safe-DS library (Group 2). If someone could not solve a step on their own, which happened *40 times* in Group 1 and twice in Group 2, we used the mean time that other participants who did solve it required. Lower values are better. 64

Figure 27: Defining a DS pipeline in Altair AI Studio. Boxes represent individual operations. Half-circles (ports) stand for inputs on the left of boxes, and for outputs on the right. The color of ports visualizes their type. Output and input ports of compatible type can be connected via edges. 71

Figure 28: Parameters of the `Discretize` operation that can be set in a sidebar. Input fields are chosen based on the type of the parameter and metadata about the values connected to input ports. 71

Figure 29: Automatically generated API reference for the `Table.fromCsvFile` method from Listing 21. Types are clickable and jump to the corresponding part of the API reference. Usages of the documented API element in examples are highlighted. 86

Figure 30: Inferring the type of a reference by traversing the AST until we find a manifest type. . . 96

Figure 31: Some type rules for concepts of the Safe-DS language. The judgments above the line are the *premises*, the judgment below the line is the *conclusion*. 97

Figure 32: Inferring the value of a placeholder by traversing the AST until we find a literal. 99

Figure 33: The *hidden state* of a notebook maintains a mapping from variables to their current value. Assignments update the internal state and references look up the current value. 105

Figure 34: An example notebook with three code cells. It contains Python code that calls functions from the Safe-DS library (Part II). 106

Figure 35: A *dependency graph* for the *cells* in Figure 34. An arrow from cell `A` to cell `B` indicates that `B` depends on `A`. We also say that `B` is *downstream* of `A`, or `A` is *upstream* of `B`. 108

Figure 36: A basic dependency graph for the *operations* in Figure 34. Boxes stand for calls, and the 💾 symbol for variables. An arrow to the 💾 symbol indicates an assignment to a variable. An arrow from

the 💾 symbol indicates a reference to a variable. Arrows between boxes show chaining or nesting of calls. An arrow from operation `A` to operation `B` indicates that `B` depends on `A`. 109

Figure 37: An enhanced dependency graph for operations in Figure 34 that also captures file interactions. The 📄 symbol stands for a file, an arrow from the 📄 symbol indicates a file read and an arrow to the 📄 symbol is a file write. 110

Figure 38: The plan for partial execution until the `cleaned` variable is easily derived from Figure 37. All upstream operations must be run, all others can be skipped. Upstream operations must be run first. 111

Figure 39: The plan for partial execution to update all files. Again, all upstream operations must be run, all others can be skipped. Upstream operations must be run first. 112

Figure 40: Comparison of a naive and memoized computation of the fourth Fibonacci number. ... 112

Figure 41: An augmented dependency graph with information about the purity of functions. Impure functions with side effects are red (`to_csv_file`), impure functions without side effects are yellow (`from_csv_file`), pure functions are green (the others). 113

Figure 42: The IDE and runner communicate via WebSockets (green). The runner has a main process, which only handles WebSockets communication and delegates the actual program execution to worker processes. Communication between the main and worker processes happens via a message queue (purple). All processes can access the memoization table (blue) via shared memory. 120

Figure 43: Statistics shown by Datalore for various types and distributions of data. Potential problems with data quality get highlighted. Some statistics for the `age` column were hidden for brevity. 131

Figure 44: Data Wrangler has a combined view for statistics and data and can edit data. 132

Figure 45: Graphical tools for DS pipelines have dedicated GUIs for value inspection. Orange and KNIME integrate results into the dialog for changing primitive arguments of an operation. 132

Figure 46: An interactive dashboard created by Power BI. Clicking on plot elements (like the `"female"` slice in the pie chart) selects the corresponding portion of the data and updates other plots. 133

Figure 47: The abstract syntax tree (AST, yellow) connects views onto a pipeline (blue) via unidirectional or bidirectional links. Tabular data that is computed by running the generated Python code is presented in the table explorer. Static checks are implemented once using the AST. 135

Figure 48: Diagnostics are shown directly in the code as it is being written. Markers are localized precisely to the faulty code. Messages provide potential resolutions, where possible. 135

Figure 49: Code-completion shows all API elements that can be accessed in the current context (here the members of the `Table` class). Entering text filters the suggestions using fuzzy matching. The description for the selected entry is shown in an additional panel. 136

Figure 50: Snippets insert interactive templates into the code. They contain filler texts that should be changed by users (here the name, arguments, and body of the segment). Users can navigate between the filler texts with the tab key. 136

Figure 51: Basic syntax highlighting vs. semantic highlighting. The latter is run after reference resolution, so it can style e.g. references to methods differently from references to classes. 137

Figure 52: Tooltips are shown when hovering over declarations and references and render the corresponding documentation comment. 137

Figure 53: Inlay hints show inferred information inside the code, like the type of a placeholder, or the name of the parameter that corresponds to a positional argument. 137

Figure 54: The outline view shows the main symbols in the current file, like the package (`demo`), pipelines (`whoSurvived`), and segments (`preprocessTitanic`). Symbols that contain diagnostics are marked (e.g. `preprocessTitanic`, which has one warning). 138

Figure 55: Searching for references to a symbol reveals its declaration and all places that use it, which works across files. Clicking on an entry jumps to the location in the code. 138

Figure 56: Values of placeholders can be inspected via *code lenses*, which are clickable actions in the code. Code lenses are displayed automatically based on the inferred type of a placeholder. They trigger partial execution (Section 31.2) and open an appropriate view for computed value. 138

Figure 57: A generic expression node for any expression without dedicated graphical syntax. Its value is accessed by an output port, which is the small circle on the right. 139

Figure 58: A call node with input ports on the left for parameters and output ports on the right for results. Input ports are filled with white for required parameters and with gray for optional one. Ports for required parameters that are not set have a red border to highlight this error. 140

Figure 59: Two chained method calls. Edges going into the input ports pass arguments. Edges coming out of output ports access that result. The receiver is modeled as another input port. 140

Figure 60: Literal arguments can be viewed and edited in a sidebar to reduce clutter in the graph. It is opened by clicking on a call node. Here, the node for `fromCsvFile` is selected (indicated by the blue border). Input ports are filled with black to denote that their value is entered in the sidebar. 140

Figure 61: Assigning the result for the call to `removeColumns` to the placeholder `titanic`. Assignment is indicated by an edge that ends at the input port (left side) of a placeholder node. The value of a placeholder can be retrieved by the output port (right side) of the node. 141

Figure 62: External view of a segment call and internal view of its implementation. In the internal view, parameters and results are still shown by input ports (left) and output ports (right) respectively. But now, edges *start* at input ports (parameter reference) and *end* at output ports (`yield` of result). 141

Figure 63: Lambdas get inlined into the input port for callbacks of higher-order functions. Clicking on the port reveals the implementation of the lambda. 142

Figure 64: A toolbox to add nodes to the graph. Entries are grouped by category and can be searched. Selecting an output port in the graph keeps only the entries with compatible input ports. 143

Figure 65: The main panel of the table explorer shows the data as well as statistics, visualizations, and quality metrics for each column. Potential quality issues are highlighted in red. 143

Figure 66: Clicking on the funnel icon opens a filter dialog. Its input fields are determined by the type and distribution of the column. Since `age` is numeric, users filter values using a range. 144

Figure 67: In the plot editor, users pick a plot type and the columns to visualize in dropdowns. For this figure, the actual plot has been shrunk in relation to the dropdowns to save space. 145

Figure 68: The history shows all actions done in the table explorer that change data. The current step is printed in bold, and steps can be undone or redone. 145

Figure 69: A screenshot of the work-in-progress implementation of the graphical view. 146

Figure 70: A box plot of the SUS scores for all groups. Higher values are better. 149

Figure 71: A box plot of the SUS-like scores for learnability for all groups. It is based on two items of the normal SUS questionnaire, and a custom one. The final score is between 0 and 100. Higher values are better. 149

Figure 72: A box plot of the time (in minutes) to solve the task in all groups. If someone could not solve a step on their own, we used the mean time that other participants who did solve it required. Lower values are better. 150

Listings

Listing 1: Python docstrings with a general description, a parameter description, and a parameter type in popular formats. 14

Listing 2: The signature of the original function. Its implementation is omitted. 15

Listing 3: Initial, trivial adapter with the original signature. The docstring is omitted. 15

Listing 4: Adapter after removing the default value of `indent`. Note that the code is invalid, since the optional parameter `encoding` occurs before the required parameter `indent`. This is fixed next. 16

Listing 5: Adapter after reordering parameters during postprocessing. 16

Listing 6: Problems with static usage analysis. 20

Listing 7: Description of parent-child links in the AST for Python implemented in Kotlin. 22

Listing 8: The core structure of a Safe-DS pipeline file. It first states its *package* (line 1), which is the namespace for all declarations in the file. Then, declarations from other packages are *imported* (line 3). Finally, a *pipeline* is defined (line 5), which is the entry point of a program. 75

Listing 9: Statements define the actions of a pipeline. An *assignment* (line 6) stores the results of an expression in *placeholders*. An *expression statement* (lines 8–10) executes an expression for its side effects, like writing data to a file but discards its results. 75

Listing 10: Calling higher-order functions with *lambdas* as arguments. *Block lambdas* (lines 12–15) can contain multiple statements and specify their results using assignments and the `yield` keyword. *Expression lambdas* (lines 17–19) are syntactic sugar and compute a single expression. 77

Listing 11: Common code can be extracted into a *segment* and called multiple times in a pipeline. 78

Listing 12: A stub file states its *package* (line 1), its imports (line 3) and finally a list of API element descriptions. Lines 5–7 show the start of a class description that is continued in Listing 13. 79

Listing 13: A class with a constructor (line 5), an instance attribute (line 6), and a static method (lines 8–11) with a required parameter (line 9), an optional parameter (line 10), and a single result (line 11). Attributes, parameters, and results have manifest types. 80

Listing 14: A global function with a required parameter of type `String` and a result of type `Int`. . . 81

Listing 15: An *enumeration* defines a finite, fixed set of *variants*. Each variant has its own list of parameters. Each parameter automatically defines an attribute for the variant with the same name and type. 81

Listing 16: A Python class that corresponds to the Safe-DS class from Listing 13. For brevity, the implementations of the methods are omitted. 82

Listing 17: Python classes that correspond to the Safe-DS enumeration from Listing 15. 83

Listing 18: The `@PythonName` annotation allows using API elements of a Python library under a different name in the pipeline language. 83

Listing 19: The `@PythonModule` annotation decouples the structure of Safe-DS packages from Python modules. It affects all stubs in the file. 84

Listing 20: With the `@PythonMacro` annotation, calling a function in the Safe-DS language can run an arbitrary Python expression. 84

Listing 21: Documenting declarations in the stub language using a JavaDoc-like format. The comment starts with a description of the documented method (line 4). Its parameters and results are then documented individually by using the `@param` and `@result` tags, followed by the name of the documented parameter and result. Examples are added using the `@example` tag. 84

Listing 22: The `@Deprecated` annotation stipulates that an API element should no longer be used and can suggest alternatives. 85

Listing 23: Type of lambda parameters are inferred from the manifest type of the corresponding parameter in a callable type. 97

Listing 24: A class with a *boundary constraint*. The parameter `treeCount` only accepts values larger than `0`. The `const` modifier ensures that only values are passed that can be evaluated to a constant. 100

Listing 25: Tracking whether a classifier is fitted yet using an enumeration and a type parameter. The constructor of `Classifier` returns an unfitted classifier, while the `fit` method returns a fitted one. 100

Listing 26: A *state constraint* that only allows calling `predict` on a fitted classifier. Otherwise, a custom error message is shown. 101

Listing 27: A *schema constraint* that checks whether a table has some column. 102

Listing 28: Safe-DS stubs with purity annotations. The method `fromCsvFile` is impure because it reads from a file (given by its `path` parameter), but `shuffleRows` is pure. 116

Listing 29: Inferring the purity of the `mystery` call by building a call graph with purity information. Green nodes are pure functions, yellow nodes read from a file. 116

Listing 30: Block lambdas do not exist in Python, so we must instead translate them to a nested function definition and a function pointer. 117

Listing 31: A call to a Safe-DS function with the `@PythonMacro` annotation is translated to Python by substituting template expressions in its template. 117

Listing 32: The code generator tells the runner (Section 32.4) which calls can be memoized. For statically dispatched functions, the code generator includes the unique qualified name of the function (line 2), a pointer to the function to call (line 3), the positional arguments (line 4), the named arguments (line 5), and the hidden arguments (line 6). 118

Listing 33: The `@Category` annotation accepts a variant of the `PipelineStep` enumeration. This information is used to group related entries in the toolbox together. 143

Tables

Table 1: Approaches towards a new API for a library. 7

Table 2: Wordings for boundaries from the documentation of scikit-learn. 13

Table 3: Conditions of parameter dependencies from the documentation of scikit-learn. 13

Table 4: Actions of parameter dependencies from the documentation of scikit-learn. 13

Table 5: Application of inferred transformations on adapters. 16

Table 6: Application of manually applied transformations on adapters. 18

Table 7: Inferred optionality of useful parameters of scikit-learn with $\alpha = 0.05$. Cases where optionality or the default value changed are highlighted. 25

Table 8: Precision of precondition extraction. The numbers in the table refer to the full library. 25

Table 9: Recall of precondition extraction. The numbers in the table refer to a random sample of 200 functions of each library. 25

Table 10: General information about popular Python libraries for DS from October 28, 2024. The column "Regular use" indicates the percentage of respondents to [102] that regularly use a library. The data for stars, contributors, and downloads only includes the main repository and package. 34

Table 11: The libraries we picked as basis for the Safe-DS library. 56

Table 12: The analyzed tools for improving safety when developing DS pipelines. 72

Table 13: The three rules to build terms in the lambda calculus. 78

Table 14: Is the non-nullable type A (rows) a subtype of the non-nullable type B (columns)? Cases with a ✓ are always true, cases with a ✗ are always false, and cases with a ? are discussed further in this section. "Other class" stands for any class type but `Nothing` and `Any`. 92

Table 15: Converting value options of a literal type to a class type. 93

Table 16: Places where a type parameter may be used and their base position. We assign a numeric multiplier to each kind of position (in = −1, invariant = 0, out = 1). The position of the type parameter can then be determined by traversing the AST and aggregating all multipliers. 95

Table 17: The extended memoization table after running the execution plan from Figure 38. For each call, it stores the called function, the arguments of the call, and its results. Impure functions that only *read* from files can be memoized by adding last modified timestamps of the accessed files. Identical colors for argument and result tables indicate the same object is referenced. 114

Table 18: Aggregated runtimes of the example notebook (Figure 34) in milliseconds using our execution approach for Safe-DS, and when running it with marimo in various cell configurations. 122

Table 19: Effect of memoization on the runtime (in seconds) of Kaggle pipelines during the first run and repeated runs. The table also lists the size of the input files on disk and the size of the memoization table in memory in Megabyte. Large improvements due to memoization are highlighted. 123

Table 20: Effect of strategies for freeing memory by deleting entries from the memoization table on pipeline runtimes (in seconds). Full memoization (last column) is best in all cases, but would not match the memory constraint. The next best option that fits into memory is marked. 124