Inspect4py: A Knowledge Extraction Framework for Python Code Repositories

Rosa Filgueira
University of St Andrews
St Andrews, UK
rf208@st-andrews.ac.uk

Daniel Garijo
Universidad Politécnica de Madrid
Madrid, Spain
daniel.garijo@upm.es

ABSTRACT

This work presents **inspect4py**, a static code analysis framework designed to automatically extract the main features, metadata and documentation of Python code repositories. Given an input folder with code, inspect4py uses abstract syntax trees and state of the art tools to find all functions, classes, tests, documentation, call graphs, module dependencies and control flows within all code files in that repository. Using these findings, inspect4py infers different ways of invoking a software component. We have evaluated our framework on 95 annotated repositories, obtaining promising results for software type classification (over 95% F1-score). With inspect4py, we aim to ease the understandability and adoption of software repositories by other researchers and developers.

Code: https://github.com/SoftwareUnderstanding/inspect4py

DOI: https://doi.org/10.5281/zenodo.5907936

License: Open (BSD3-Clause)

CCS CONCEPTS

• **General and reference → Surveys and overviews**; • **Applied computing → Document capture**; **Document analysis**.

KEYWORDS

Code mining, software code, software classification, software documentation, code understanding

ACM Reference Format:

Rosa Filgueira and Daniel Garijo. 2022. Inspect4py: A Knowledge Extraction Framework for Python Code Repositories. In *Proceedings of MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

An increasing number of research results depend on software, in areas ranging from High-Energy Physics [15] to Computational Biology [12]. Research software is used to clean and analyze data, simulate real systems or visualize scientific results [3].

In the last years, research software has become a subject of interest for the scientific community for two main reasons. First, software itself has become a research topic, with multiple research challenges such as using efficient code representations [14] for function similarity [7] or generating documentation from code [4]. Second, the importance of software for Science has promoted the worldwide adoption of the Findable, Accessible, Interoperable and Reusable principles (FAIR) [17] adapted to software [1]. However, to this date software reuse is still a time-consuming task [11].

This work presents **inspect4py**, a Python static code analysis framework designed to 1) extract the most relevant information from all files contained in a software repository (such as functions, classes, methods, documentation, dependencies, call graphs and control flow graphs), and 2) use the extracted information to classify the type of a code repository (i.e, package, library, script or service), detecting alternative ways of running it.

Our framework aims to facilitate the creation of different code feature representations; and ease code repository comprehension [16]. We have validated our approach with an annotated corpus from 95 Python repositories, which shows promising results (over 95% F1-score). All extracted results are stored locally in JSON and HTML formats, making it easy to consume them in human-readable and machine readable manner.

The rest of the paper is structured as follows. Section 2 presents an overview of the framework, while Section 3 describes our evaluation. Section 4 describes related work and tools, Section 5 shows execution examples and Section 6 concludes the paper.

2 INSPECT4PY

Inspect4py is a standalone Python 3 package developed to ease software adoption and analysis. Given a Python code repository, inspect4py extracts relevant code features, detects class and function metadata, and identifies the main entry points for using it. This section overviews the main features of our framework, and provides insight into their rationale and design.

2.1 Overview

Inspect4py takes as input a code repository folder, and extracts two main types of features:

Software understanding features, aimed at easing the adoption of a software package:

- **Class and function metadata and documentation**: for each of the classes in the code repository, inspect4py will retrieve their name, inherited classes, documentation and respective methods. For each function, our framework detects its arguments, documentation, returned value, relevant variables which store other function calls and whether the function is nested (i.e., it defines further functions within).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MSR 2022, May 23–24, 2022, Pittsburgh, PA, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

- **Requirements**: list of packages needed to run the target software, along with their corresponding version.
- **Tests**: list of files that are used to test the functionality of a software component (and usually not relevant for using it).
- **Software invocation**: a ranked list with the different alternatives to run a target software component, ordered by relevance.
- **Main software type**: an estimation on whether the target software is aimed to be a package, library, service or a series of scripts.

Code features, aimed at characterising code under different perspectives:

- **File metadata**: for each file, we track its included classes and methods, together with its dependencies (imported modules), whether there is a main method declared, or whether the file has a *body* (i.e., files without a *main* function, but with calls to functions/methods and/or instantiating classes).
- **Control flow graph**: for each file, we retrieve its control flow representation as a text file and figure (png, dot or pdf). The control flow graph contains alternative insights on the possible paths followed for executing a program.
- **Call graph**: for each function (or body of code) we extract a call graph of all involved functions, including those passed by argument, by assignment or by dynamic means.
- **File hierarchy**: we record how different files have been grouped and organized in a software repository.

As a result, inspect4py produces a folder containing a summary JSON file (*directory_info.json*) with the features selected by users. A sub-folder is also created for each of the code files in the original repository, including JSON results at an individual level, and control flow results. The following sections elaborate on how these features have been implemented.

2.2 Extracting class and function metadata

Following common practices in static code analysis [10], inspect4py uses Abstract Syntax Trees (ASTs) [9] to obtain the full representation of all code in a software repository. In particular, we use the ast.walk()[1] function, which recursively yields all descendant nodes in a code tree.

Inspect4py uses different *ast* classes (e.g *FunctionDef*, *Call*, *Assign*, *Return*, etc.) to extract automatically all the details of classes, methods, functions, and their respective documentation. This method allows capturing the relevant nodes of the tree in memory for further manipulation and specific analysis, e.g., when detecting concrete function names like *main*, navigating through the imported dependencies or assessing whether a function is a test or not.

Listing 1: Sample Python script (Example.py)

```
import os
path=os.path.join("/User", "/home", "file.txt")
def width():
    return 5
def area(length, func):
    print(length * func())
area(5, width)
```

[1]https://docs.python.org/3/library/ast.html?highlight=walk#ast.walk

functions	**area**	**local**	• print • Example.width
body	**local**	• os.path.join • Example.area	

Figure 1: Call graph obtained for *Example.py* (Listing 1)

2.3 Call graph and control flow

Using the parsed AST, inspect4py creates a call graph for each function, method or *body* in each code file. The call graph includes all the functions invoked within that function, method or *body*. We support four main types of Python functions:

- **Direct functions**, referred directly by their name in code.
- **Argument functions**, passed as an argument/parameter of another function.
- **Assignment functions**, assigned to a variable, which is then used to make the function call.
- **Dynamic functions**, i.e., determined only at run time by its input parameter of another function call.

An example of our results can be seen in Figure 1, which depicts the HTML representation of the JSON call graph obtained after parsing the Python script in Listing 1.

The control flow representation of each file is achieved through the cdmcfparser[2] and staticfg[3] tools, which create a hierarchical representation of a code file in fragments.

2.4 Module requirements and dependencies

Understanding which are the requirements and dependencies of a software component is crucial to ease its installation and assess any security issues. Inspect4py captures this information at two levels of granularity:

- **File level**, by inspecting in the AST which modules are imported within each file of a code repository.
- **Repository level**: We have incorporated Pigar,[4] an open Python package designed to generate the list of requirements needed to run a given Python repository. We selected Pigar for its ability to handle different Python versions (2.7+ and 3.2+), retrieve the version of an installed package, and its support for Jupyter notebooks. Internally, Pigar also uses *ASTs* instead of regular expressions, and is able to extract *import* statements from *exec* or *eval* expressions, parameters and document strings.

2.5 Main software type and invocation

Knowing how to easily invoke a code repository is crucial for its adoption. This information is usually available in README files, but less frequently in a machine readable format. Inspect4py aims to address this issue by automatically detecting which software type a target code repository is (package, library, service or script); and by finding alternative ways of executing it.

[2]https://github.com/SergeySatskiy/cdm-flowparser
[3]https://github.com/coetaur0/staticfg
[4]https://github.com/damnever/pigar

2.5.1 *Software types.*

We distinguish four main types of software:

- **Package**: tools that are aimed to be executed through the command line.
- **Library**: codes which are aimed to be imported as part of other software.
- **Service**: codes where the main functionality is to start a web service, usually through a script.
- **Script**: codes that do not fit any of the previous categories, and are run with in one or several executable files (either through a main function, or a script with *body*).

While many Python repositories may be classified in only one of these categories, it is common to find many code repositories with an overlap. For example, a package may be imported as a library, or a library may also have a demo script showing how to use it in detail. Therefore, inspect4py returns the *main* estimated functionality of the target repository, together with a ranked list of alternatives for running it.

2.5.2 *Software invocation and type classification.*

We have created a heuristic based on best practices for packaging Python projects[5] and the analysis of all code files in a target repository. We focus on metadata files, both dynamic (*setup.py*) and static (*setup.cfg*), files with a *main* function, and files with *body*.

Our heuristic first looks whether the repository has a metadata file ('setup.py' or 'setup.cfg' files (or both)) or not, as these files usually contain useful information about the code repository itself. If found, we look for: 1) the software project name; and 2) entry points, which are typically console scripts, functions or other callable function-like objects identified by developers to make available as a command-line tools.

If entry points are found, we check whether any console scripts have a significant overlap with the name of the software component. The rationale for this is that, in most observed cases, package names and console entry points are very similar. If there is significant overlap, we mark the software repository as a **package**. If not, or if there are no entry points, we consider the code repository to be a **library**. Next, we analyze the rest of the files of the repository individually:

Service files: Using the information stored in the AST, we check if the modules and libraries imported in the analyzed file correspond to any of the major libraries commonly used for creating web services. [6] If so, we mark the file as a **service** script.

Test files: Many software projects include test files to ensure new updates do not alter the expected behavior of the developed application. These files are crucial in application development, but less important to understand and adopt a software component. We detect tests by reviewing whether a testing framework is included in the file dependencies,[7] and by assessing whether most functions in the file use the *assert* function, commonly used when testing applications. If a test file has a *main* function, we annotate it in our results as well.

Script files: If a file is not tagged as a test or a service, we check whether it is an executable file or not. As described above, we use the AST of the file to identify *main* functions within each file, as well as those executable files with no main functions, but with a *body*. In both cases, we tag the file as a **script**.

We perform an additional analysis among scripts with main function to provide additional insight: we explore their call graph using a depth-first search and identify their dependencies (direct or indirect) with other scripts with main function. In case a script is imported by another script, it will be marked as *secondary*. For each script, our results also store its full list of imported scripts.

2.5.3 *Ranking software invocation methods.*

Once our file analysis finishes, we rank our invocation results by estimating their relevance. We have a scoring function, which assigns weights to each result based on the features resulting from the previous analysis. Different features have different weights, according to the common practices observed empirically by the authors in over 90 code repositories. The scoring function is defined as:

$$\begin{aligned} score(x) = w_{lib} * lib(x) + w_{readme} * readme(x) + \\ + w_{service} * service(x) + \\ + w_{main} * main(x) + w_{body} * body(x) \end{aligned} \tag{1}$$

where x denotes the invocation being scored, *lib(x)*= 1 if the code repository was identified as a library or package (zero otherwise), *service(x)* = 1 if x is identified as a service, and *main(x)* and *body(x)* equal to 1 if x has a main function or only body (they equal to zero otherwise). We also consider if a service or script is mentioned in the README file (*readme(x)* = 1) as this indicates that the service or script was identified as significant by the software authors. The weights associated with each of these features (represented with a w, like w_{lib}), are highest for package or libraries, followed by services and finally, scripts. If an executable file has a *main* function, its weight will be higher than a script with just *body*.

Listing 2: JSON snippet showing two invocation alternatives

```
"software_invocation": [
    {
      "run": [
        [
          "pylode ",
          "pylode.cli:main,"
        ]
      ],
      "type": "package",
      "installation": "pip install pyLODE",
      "ranking": 1
    },
    {
      "type": "service",
      "run": "python pyLODE/pylode/server.py",
      "has_structure": "body",
      "mentioned_in_readme": true,
      "ranking": 2
    }
  ],
  "software_type": "package"
```

Once all invocation alternatives have been assessed, we sort them by their score in descending order. To improve readability, we create a ranking in ascending order, where the first position is assigned to the entry with the highest score. The first ranked element is returned as the main software type. If two invocation alternatives have the same score, they are assigned the same ranking number. Listing 2 shows a simple example of a package where the

[5] https://packaging.python.org/en/latest/tutorials/packaging-projects/

[6] list of services: flask, flask_restful, falcon, falcon_app, aiohttp, bottle, django, fastapi, locust, pyramid, hug, eve, connexion

[7] list of test libraries: unittest, pytest, nose, nose2, doctest, testify, behave, lettuce

main invocation is through the command line, but that also has an alternative invocation as a service. Inspect4py detects the main command to run the package, (available in the *setup.cfg* found in the code repository), together with another a script (*server.py*), mentioned in the main README.

3 PRELIMINARY EVALUATION

An initial assessment of inspect4py compares software type and invocation results against a manually annotated corpus.

3.1 Annotated corpora

We have created two different corpora to assess our approach. For the main software type detection, we selected a corpus of 95 different Python code repositories (distributed in 24 packages, 27 libraries, 13 services and 31 scripts).[8] All repositories have been annotated by each of the authors based on their documentation, separately, and compared until agreement was reached. The repositories are heterogeneous in scope and domain, and range from research repositories used in Machine Learning[9] to popular community tools such as Apache Airflow[10] or domain-specific libraries (e.g., Astropy[11]). We also included repositories developed at the author home institutions, with less available documentation or in development, in order to obtain a wider representative sample.

The second corpus was designed to assess the ranking results, and it was generated as a subset of the first one. For all repositories with multiple invocation methods, the authors manually annotated the most relevant files, based on the repository structure and documentation. As a result, 44 repositories were annotated.

3.2 Results

Table 1 shows an overview of our results for main software type classification, for each of the categories supported. Our heuristics, based on best practices for Python application development, yield over 95% F1-score for all categories. The only errors we encounter occur when developers step outside the common practices, such as creating custom metadata files for their libraries.

For assessing our ranking results, we selected the normalized discounted cumulative gain (NDGC) [5], a metric used in information retrieval to assess ranking quality. NDGC ranges from 0 (min) to 1 (max). Our aggregated ranking amounts to 0.87, which is considered satisfactory for a preliminary evaluation.

4 RELATED WORK

A number of static code analysis tools have been developed for extracting code metadata, documentation or features [10]. From a Machine Learning perspective, [14] describes a survey of techniques for code feature extraction, in order to train source code models. Tools like libsa4py [8] create features from code that would complement the information already extracted in inspect4py.

[8]Benchmark is available at: https://doi.org/10.5281/zenodo.5907936
[9]https://paperswithcode.com/
[10]https://github.com/apache/airflow
[11]https://github.com/astropy/astropy

Software type	Precision	Recall	F1-score
Package	1	0.916	0.956
Library	0.93	1	0.9637
Service	1	1	1
Script	0.967	0.967	0.967

Table 1: Results for software type classification.

Other tools present some overlap with the functionality included in our framework. For example, pydeps[12] and modulegraph2[13] extract module dependencies, libraries like *pydoc*[14] generate HTML code documentation, libraries like *Pigar*[15] extract code requirements, and packages like *pyan* [2] and *pycg* [13] generate call graphs for Python code using static analysis, including high order functions, class inheritance schemes and nested function definitions. Inspect4py builds on some of these tools (e.g., pygar), integrating all their functionalities into a single framework and using a unique representation for all their results.

To the best of our knowledge, there are no frameworks for detecting software type and invocation methods.

5 INSTALLATION AND EXECUTION

Inspect4py can be installed through pip (`pip install inspect4py`) and Docker. To invoke the tool with basic functionality (i.e., extract classes, functions and their documentation), one needs to run the following command:

```
inspect4py -i <input file.py | directory>
```

This command can be qualified with different options,[16] in order to perform different functionalities. For example, the following command extracts classes, function and method documentation and also stores the software invocation information (flag *-si*):

```
inpsect4py -i repository_path -o out_path -si -html
```

6 CONCLUSIONS AND FUTURE WORK

This paper introduces inspect4py, a static code analysis framework designed to extract common code features and documentation from a Python code repository in order to ease its understanding. A preliminary evaluation shows promising results evaluating Python code repository types, as well as identifying and ranking alternatives ways of running them (thus saving time to potential users).

We are using inspect4py in combination with software metadata extraction tools [6] to suggest recommendations for completing README files.[17] In addition, we are planning to use our results for facilitating function parallelization and composition, as well as to compare alternative representations of software when finding similar code.

As for future work, we are expanding inspect4py to 1) improve our evaluation results, by annotating additional number of repositories, 2) incorporating new feature extraction tools (e.g., libsa4py), and 3) improving our invocation detection to include illustrative parameter examples.

[12]https://github.com/thebjorn/pydeps
[13]https://pypi.org/project/modulegraph2/
[14]https://docs.python.org/3/library/pydoc.html
[15]https://github.com/damnever/pigar
[16]See https://github.com/SoftwareUnderstanding/inspect4py/blob/main/README.md
[17]https://github.com/SoftwareUnderstanding/completeR

REFERENCES

[1] Neil P. Chue Hong, Daniel S. Katz, Michelle Barker, Anna-Lena Lamprecht, Carlos Martinez, Fotis E. Psomopoulos, Jen Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, Tom Honeyman, Alexander Struck, Allen Lee, Axel Loewe, Ben van Werkhoven, Catherine Jones, Daniel Garijo, Esther Plomp, Francoise Genova, Hugh Shanahan, Joanna Leng, Maggie Hellström, Malin Sandström, Manodeep Sinha, Mateusz Kuzak, Patricia Herterich, Qian Zhang, Sharif Islam, Susanna-Assunta Sansone, Tom Pollard, Dwi Atmojo; Udayanto, Alan Williams, Andreas Czerniak, Anna Niehues, Anne Claire Fouilloux, Bala Desinghu, Carole Goble, Céline Richard, Charles Gray, Chris Erdmann, Daniel Nüst, Daniele Tartarini, Elena Ranguelova, Hartwig Anzt, Ilian Todorov, James McNally, Javier Moldon, Jessica Burnett, Julián Garrido-Sánchez, Khalid Belhajjame, Laurents Sesink, Lorraine Hwang, Marcos Roberto Tovani-Palone, Mark D. Wilkinson, Mathieu Servillat, Matthias Liffers, Merc Fox, Nadica Miljković, Nick Lynch, Paula Martinez Lavanchy, Sandra Gesing, Sarah Stevens, Sergio Martinez Cuesta, Silvio Peroni, Stian Soiland-Reyes, Tom Bakker, Tovo Rabemanantsoa, Vanessa Sochat, and Yo Yehudi. 2021. FAIR Principles for Research Software (FAIR4RS Principles). (2021). https://doi.org/10.15497/RDA00065 Publisher: Research Data Alliance.

[2] D. Fraser, E. Horner, J. Jeronen, and P. Massot. 2020. Pyan3: Offline call graph generator for Python 3. https://github.com/davidfraser/pyan. Accessed: 09-January-2022.

[3] Morane Gruenpeter, Daniel S. Katz, Anna-Lena Lamprecht, Tom Honeyman, Daniel Garijo, Alexander Struck, Anna Niehues, Paula Andrea Martinez, Leyla Jael Castro, Tovo Rabemanantsoa, Neil P. Chue Hong, Carlos Martinez-Ortiz, Laurents Sesink, Matthias Liffers, Anne Claire Fouilloux, Chris Erdmann, Silvio Peroni, Paula Martinez Lavanchy, Ilian Todorov, and Manodeep Sinha. 2021. Defining Research Software: a controversial discussion. https://doi.org/10.5281/zenodo.5504016

[4] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[5] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated Gain-Based Evaluation of IR Techniques. *ACM Trans. Inf. Syst.* 20, 4 (oct 2002), 422–446. https://doi.org/10.1145/582415.582418

[6] Aidan Kelley and Daniel Garijo. 2021. A Framework for Creating Knowledge Graphs of Scientific Software Metadata. *Quantitative Science Studies* (11 2021), 1–37. https://doi.org/10.1162/qss_a_00167 arXiv:https://direct.mit.edu/qss/article-pdf/doi/10.1162/qss_a_00167/1971225/qss_a_00167.pdf

[7] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (oct 2019), 28 pages. https://doi.org/10.1145/3360578

[8] Amir M. Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 585–589. https://doi.org/10.1109/MSR52588.2021.00079

[9] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. *SIGSOFT Softw. Eng. Notes* 30, 4 (may 2005), 1–5. https://doi.org/10.1145/1082983.1083143

[10] Jernej Novak, Andrej Krajnc, and Rok Žontar. 2010. Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO*. 418–422.

[11] Jeffrey M Perkel. 2020. Challenge to scientists: does your ten-year-old code still run? *Nature* 584, 7822 (2020), 656–659.

[12] Andreas Prlić and Hilmar Lapp. 2012. The PLOS computational biology software section. *PLoS Computational Biology* 8, 11 (2012), e1002799.

[13] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[14] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. 2021. A Survey on Machine Learning Techniques for Source Code Analysis. arXiv:2110.09610 [cs.SE]

[15] The HEP Software Foundation, Johannes Albrecht, Antonio Augusto Alves, Guilherme Amadio, Giuseppe Andronico, Nguyen Anh-Ky, Laurent Aphecetche, John Apostolakis, Makoto Asai, Luca Atzori, Marian Babik, Giuseppe Bagliesi, Marilena Bandieramonte, Sunanda Banerjee, Martin Barisits, Lothar A. T. Bauerdick, Stefano Belforte, Douglas Benjamin, Catrin Bernius, Wahid Bhimji, Riccardo Maria Bianchi, Ian Bird, Catherine Biscarat, Jakob Blomer, Kenneth Bloom, Tommaso Boccali, Brian Bockelman, Tomasz Bold, Daniele Bonacorsi, Antonio Boveia, Concezio Bozzi, Marko Bracko, David Britton, Andy Buckley, Predrag Buncic, Paolo Calafiura, Simone Campana, Philippe Canal, Luca Canali, Gianpaolo Carlino, Nuno Castro, Marco Cattaneo, Gianluca Cerminara, Javier Cervantes Villanueva, Philip Chang, John Chapman, Gang Chen, Taylor Childers, Peter Clarke, Marco Clemencic, Eric Cogneras, Jeremy Coles, Ian Collier, David Colling, Gloria Corti, Gabriele Cosmo, Davide Costanzo, Ben Couturier, Kyle Cranmer, Jack Cranshaw, Leonardo Cristella, David Crooks, Sabine Crépé-Renaudin, Robert Currie, Sünje Dallmeier-Tiessen, Kaushik De, Michel De Cian, Albert De Roeck, Antonio Delgado Peris, Frédéric Derue, Alessandro Di Girolamo, Salvatore Di Guida, Gancho Dimitrov, Caterina Doglioni, Andrea Dotti, Dirk Duellmann, Laurent Duflot, Dave Dykstra, Katarzyna Dziedziniewicz-Wojcik, Agnieszka Dziurda, Ulrik Egede, Peter Elmer, Johannes Elmsheuser, V. Daniel Elvira, Giulio Eulisse, Steven Farrell, Torben Ferber, Andrej Filipcic, Ian Fisk, Conor Fitzpatrick, José Flix, Andrea Formica, Alessandra Forti, Giovanni Franzoni, James Frost, Stu Fuess, Frank Gaede, Gerardo Ganis, Robert Gardner, Vincent Garonne, Andreas Gellrich, Krzysztof Genser, Simon George, Frank Geurts, Andrei Gheata, Mihaela Gheata, Francesco Giacomini, Stefano Giagu, Manuel Giffels, Douglas Gingrich, Maria Girone, Vladimir V. Gligorov, Ivan Glushkov, Wesley Gohn, Jose Benito Gonzalez Lopez, Isidro González Caballero, Juan R. González Fernández, Giacomo Govi, Claudio Grandi, Hadrien Grasland, Heather Gray, Lucia Grillo, Wen Guan, Oliver Gutsche, Vardan Gyurjyan, Andrew Hanushevsky, Farah Hariri, Thomas Hartmann, John Harvey, Thomas Hauth, Benedikt Hegner, Beate Heinemann, Lukas Heinrich, Andreas Heiss, José M. Hernández, Michael Hildreth, Mark Hodgkinson, Stefan Hoeche, Burt Holzman, Peter Hristov, Xingtao Huang, Vladimir N. Ivanchenko, Todor Ivanov, Jan Iven, Brij Jashal, Bodhitha Jayatilaka, Roger Jones, Michel Jouvin, Soon Yung Jun, Michael Kagan, Charles William Kalderon, Meghan Kane, Edward Karavakis, Daniel S. Katz, Dorian Kcira, Oliver Keeble, Borut Paul Kersevan, Michael Kirby, Alexei Klimentov, Markus Klute, Ilya Komarov, Dmitri Konstantinov, Patrick Koppenburg, Jim Kowalkowski, Luke Kreczko, Thomas Kuhr, Robert Kutschke, Valentin Kuznetsov, Walter Lampl, Eric Lancon, David Lange, Mario Lassnig, Paul Laycock, Charles Leggett, James Letts, Birgit Lewendel, Teng Li, Guilherme Lima, Jacob Linacre, Tomas Linden, Miron Livny, Giuseppe Lo Presti, Sebastian Lopienski, Peter Love, Adam Lyon, Nicolò Magini, Zachary L. Marshall, Edoardo Martelli, Stewart Martin-Haugh, Pere Mato, Kajari Mazumdar, Thomas McCauley, Josh McFayden, Shawn McKee, Andrew McNab, Rashid Mehdiyev, Helge Meinhard, Dario Menasce, Patricia Mendez Lorenzo, Alaettin Serhan Mete, Michele Michelotto, Jovan Mitrevski, Lorenzo Moneta, Ben Morgan, Richard Mount, Edward Moyse, Sean Murray, Armin Nairz, Mark S. Neubauer, Andrew Norman, Sérgio Novaes, Mihaly Novak, Arantza Oyanguren, Nurcan Ozturk, Andres Pacheco Pages, Michela Paganini, Jerome Pansanel, Vincent R. Pascuzzi, Glenn Patrick, Alex Pearce, Ben Pearson, Kevin Pedro, Gabriel Perdue, Antonio Perez-Calero Yzquierdo, Luca Perrozzi, Troels Petersen, Marko Petric, Andreas Petzold, Jónatan Piedra, Leo Piilonen, Danilo Piparo, Jim Pivarski, Witold Pokorski, Francesco Polci, Karolos Potamianos, Fernanda Psihas, Albert Puig Navarro, Günter Quast, Gerhard Raven, Jürgen Reuter, Alberto Ribon, Lorenzo Rinaldi, Martin Ritter, James Robinson, Eduardo Rodrigues, Stefan Roiser, David Rousseau, Gareth Roy, Grigori Rybkine, Andre Sailer, Tai Sakuma, Renato Santana, Andrea Sartirana, Heidi Schellman, Jaroslava Schovancová, Steven Schramm, Markus Schulz, Andrea Sciabà, Sally Seidel, Sezen Sekmen, Cedric Serfon, Horst Severini, Elizabeth Sexton-Kennedy, Michael Seymour, Davide Sgalaberna, Illya Shapoval, Jamie Shiers, Jing-Ge Shiu, Hannah Short, Gian Piero Siroli, Sam Skipsey, Tim Smith, Scott Snyder, Michael D. Sokoloff, Panagiotis Spentzouris, Hartmut Stadie, Giordon Stark, Gordon Stewart, Graeme A. Stewart, Arturo Sánchez, Alberto Sánchez-Hernández, Anyes Taffard, Umberto Tamponi, Jeff Templon, Giacomo Tenaglia, Vakhtang Tsulaia, Christopher Tunnell, Eric Vaandering, Andrea Valassi, Sofia Vallecorsa, Liviu Valsan, Peter Van Gemmeren, Renaud Vernet, Brett Viren, Jean-Roch Vlimant, Christian Voss, Margaret Votava, Carl Vuosalo, Carlos Vázquez Sierra, Romain Wartel, Gordon T. Watts, Torre Wenaus, Sandro Wenzel, Mike Williams, Frank Winklmeier, Christoph Wissing, Frank Wuerthwein, Benjamin Wynne, Zhang Xiaomei, Wei Yang, and Efe Yazgan. 2019. A Roadmap for HEP Software and Computing R&D for the 2020s. *Computing and Software for Big Science* 3, 1 (Dec. 2019), 7. https://doi.org/10.1007/s41781-018-0018-8

[16] A. Von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076

[17] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* 3 (March 2016), 160018. https://doi.org/10.1038/sdata.2016.18