

FnO-Convert: Automatic Semantic Annotation of ML Pipelines Using the Function Ontology

Pol Nachtergaele¹, Ben De Meester¹ and Femke Ongenaë¹

¹IDLab, Ghent University – imec

Abstract

Transparency in machine learning (ML) is increasingly critical in sensitive domains such as healthcare and finance. Mainly, prospective provenance, i.e., a description of what a process, such as an ML script, is intended to do, is essential for trust and accountability. Existing approaches are not fully automatic, requiring either manual provenance annotations or ongoing human intervention, making them time-consuming, error-prone, and unable to capture complete script functionality. We present FnO-Convert, an open-source Python application that automatically and exhaustively extracts PROV-O compliant prospective provenance from Python scripts without requiring developer annotations, using the Function Ontology (FnO), demonstrated here on machine learning (ML) pipelines. FnO-Convert recursively converts Python scripts into hierarchical FnO descriptions spanning script structure to individual operations. To demonstrate extensibility, Dockerfile support is included, showing that FnO's separation of functional descriptions from their implementation environment allows descriptions across heterogeneous artifacts to be meaningfully linked. This demo demonstrates the FnO-Convert user interface, covering both the conversion process of the ML pipeline and the visualization of the generated RDF in a human-readable form, enabling users to interactively explore the prospective provenance of the pipeline as well as how provided data flows through it, enhancing transparency.

Keywords

Prospective provenance, Function ontology, Semantic annotation, Machine learning pipelines, PROV-O

1. Introduction

As machine learning (ML) systems are increasingly deployed in sensitive domains, such as healthcare and finance, transparency of ML scripts becomes essential [1]. A key enabler of this transparency is prospective provenance, i.e., a description of what a script is intended to do: which functions are used and how data flows between them [2]. Despite its importance, prospective provenance is rarely available because scripts remain opaque and manual annotation is costly and error-prone. [3].

Several approaches have been proposed to derive prospective provenance from codebases. YesWorkflow [4] extracts workflow structure from developer-provided annotations, while DLProv [5] captures provenance within instrumented systems based on predefined workflows. Vamsa [6] curated an extensive manually constructed semantic knowledge base of well-known machine learning APIs, which allows to generate prospective provenance of specific Python scripts without additional annotations, focusing on dataset-to-model lineage. Graph4Code [7] generates RDF graphs capturing simplified data flow and function usage across Python scripts, yet omits control flow and the internal behaviour of user-defined functions. These approaches either require developer annotations, rely on curated knowledge bases, or do not exhaustively capture the full top-down functional stack.

We present *Fno-Convert*, an open-source Python application¹ that automatically generates prospective provenance from Python scripts using the Function Ontology (FnO) [8]. The use of FnO is motivated by its separation of abstract function descriptions from implementation-specific metadata, making it extensible across languages, while its alignment with the W3C PROV Ontology (PROV-O) [9] enables direct integration with retrospective provenance descriptions. Existing approaches to describing

SEMANTiCS 2026: 22nd International Conference on Semantic Systems, September 15–17, 2026, Ghent, Belgium

✉ pol.nachtergaele@ugent.be (P. Nachtergaele); ben.demeester@ugent.be (B. D. Meester); femke.ongenaë@ugent.be (F. Ongenaë)

🆔 0009-0002-4551-1165 (P. Nachtergaele); 0000-0003-0248-0987 (B. D. Meester); 0000-0001-5198-0946 (F. Ongenaë)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

¹<https://github.com/FnoOio/fno-convert>

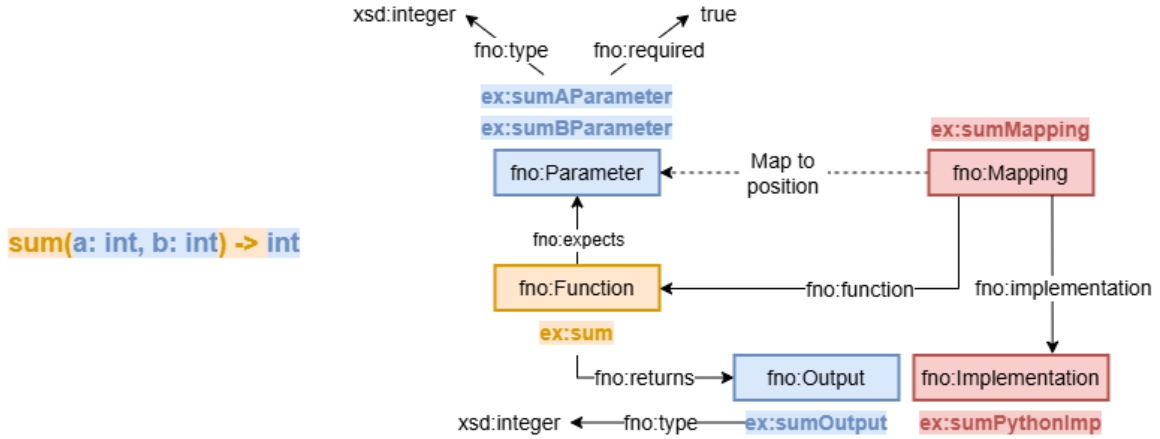


Figure 1: The core FnO components illustrated with a concrete Python example: a Function defines abstract functionality through its expected Parameters and Outputs, and is linked to its Implementation via a Mapping. At the left the figure shows the corresponding Python signature `sum(a: int, b: int) -> int`.

function semantics are typically coupled to a specific development context [10, 11, 12, 13], such as OWL-S [14] for Web services or Linked Software Dependencies [15] for software configurations. Unlike these, FnO’s implementation-independent function model allows the same abstract description to be linked across heterogeneous execution environments and languages, which we exploit here to connect Python scripts and Dockerfiles within a single provenance graph.

FnO-Convert statically analyses Python scripts to derive hierarchical descriptions from script structure down to individual operations, without manual annotations or pre-defined notions on the used ML APIs or workflows. The underlying static analysis approach is language-general and applies to any Python codebase; we focus on ML pipelines because their multi-stage structure (data loading, preprocessing, training, evaluation) and heavy reliance on complex, often loosely documented library APIs make understanding data flow between stages particularly valuable, while their deployment in sensitive domains raises the cost of remaining opaque. To demonstrate extensibility to heterogeneous artifacts, FnO-Convert also supports Dockerfiles, where individual RUN commands that reference Python scripts are linked to the FnO descriptions of those scripts, yielding a unified provenance view spanning both orchestration and execution layers. FnO-Convert maps extracted elements directly to FnO and provides an interactive visualization widget for inspecting the resulting RDF in a human-readable form.

2. Semantic Annotation of Codebases

FnO-Convert uses the FnO model to semantically annotate Python scripts (Figure 1). A Function defines abstract functionality through its expected Parameters and Outputs, each described in terms of, e.g., semantic type and optionality. An Implementation represents an executable artifact and captures how it can be retrieved (e.g., through a local file, a function defined within a file, or a function imported from an external library), and is linked to its corresponding Function via a Mapping, which specifies how parameters and outputs should be provided when executing the artifact. A Composition describes how a function is internally built from other functions, capturing data flow between their inputs and outputs. This enables descriptions ranging from input/output signatures to internal dataflow.

2.1. Semantic Annotation of Python Scripts

A Python function is semantically described by automatically inspecting its signature, extracting the function name, parameters, and their types, as shown with an example in Figure 1. This yields a Function, its Parameters and Outputs, and a corresponding Mapping. The native built-in Python inspect module is then used to generate the Implementation by capturing imported modules or local

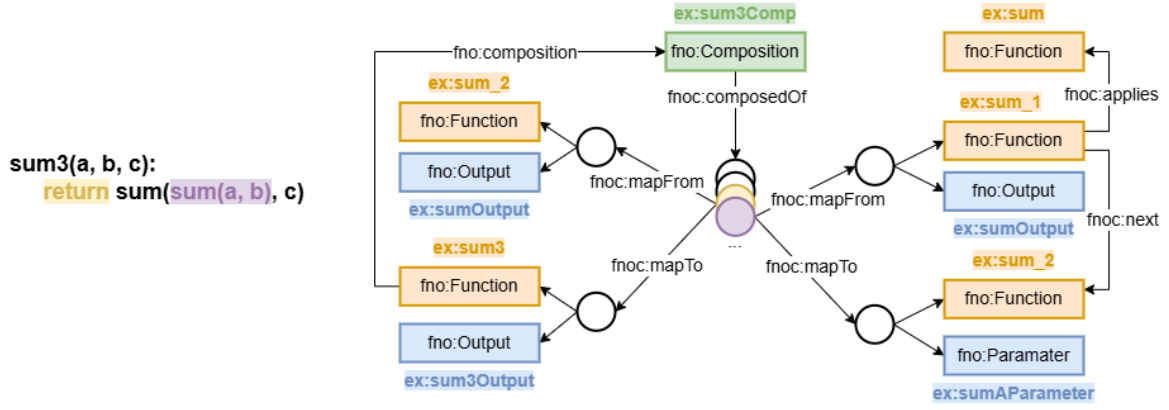


Figure 2: An FnO Composition describing the internal structure of `sum3(a, b, c)`: the function is decomposed into an ordered sequence of `sum` calls, with `fnoc:mapFrom` and `fnoc:mapTo` triples capturing how outputs of inner calls are forwarded as inputs to subsequent ones.

file paths. User-defined functions are further decomposed into a `Composition` through static analysis of their source code, as illustrated with an example in Figure 2. First, a Control Flow Graph (CFG) is constructed, capturing all potential execution paths and grouping linear statements into blocks while identifying control flow structures, such as loops and conditional branches. Each block of linear statements is parsed as an Abstract Syntax Tree (AST) to identify function calls, which are recursively described as `Functions`. Variable bindings are analysed to derive `Composition` mappings between function outputs and inputs. Control flow structures, such as loops and conditional branches, are collapsed into a single `Function` whose body is itself represented as a nested `Composition`, capturing all statically reachable execution paths while preserving execution order and keeping each composition as a linear sequence of functions. As FnO-Convert relies on static analysis, runtime-only behavior such as `eval/exec` calls and monkey-patching is not resolved and is described as written in the source. A Python file is described as a `Function` by parsing its `argparse` definitions to identify the expected inputs. The Mapping captures the invocation semantics of each input, including whether it is a positional argument or a named option, and whether it expects a single value or multiple values. FnO-Convert then inspects the code that will be executed upon invocation and recursively parses any called functions to produce a full `Composition` of the file. This recursive process enables descriptions at arbitrary levels of granularity.

2.2. Semantic Annotation of Docker Files

A Dockerfile defines a reproducible execution environment through a sequential set of instructions that build a Docker image. Using the vocabulary introduced by Tommasini et al. [16], these instructions are represented as `Functions`, allowing FnO-Convert to describe the full Dockerfile as a `Composition`. FnO-Convert inspects `RUN` commands for Python file references and extracts CLI inputs. Referenced files are recursively parsed and linked as a `Composition` of the `RUN` command's `Function`, such that the `RUN` command is internally described as an invocation of the referenced script with its specific inputs.

3. FnO-Convert Provenance Visualization & Demonstration

FnO-Convert is implemented as a PyQt6 desktop application². The source code is publicly available at <https://github.com/FnOio/fno-convert> under permissive MIT license along with a demo video. Upon startup, users are presented with the Load Tab (Figure 3). Widget 1 allows the user to load a Python file, Dockerfile, or a Turtle file containing previously generated FnO descriptions. Python and Docker files

²<https://www.riverbankcomputing.com/software/pyqt/>

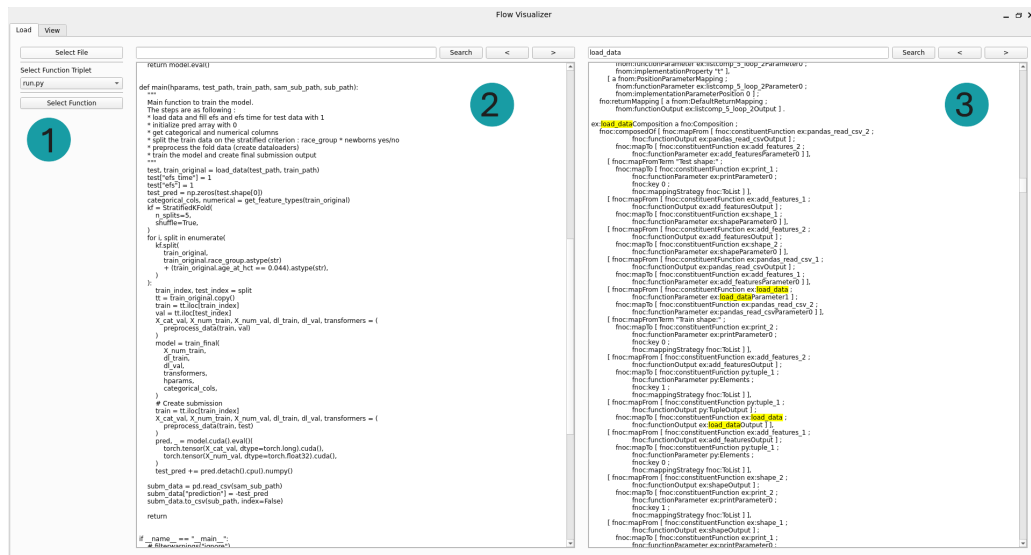


Figure 3: The FnO-Convert Load Tab: Python and Dockerfiles are automatically parsed or FnO Turtle files are loaded (Widget 1), source code is displayed (Widget 2), and Functions can be selected in Widget 1 to inspect its annotations (Widget 3).

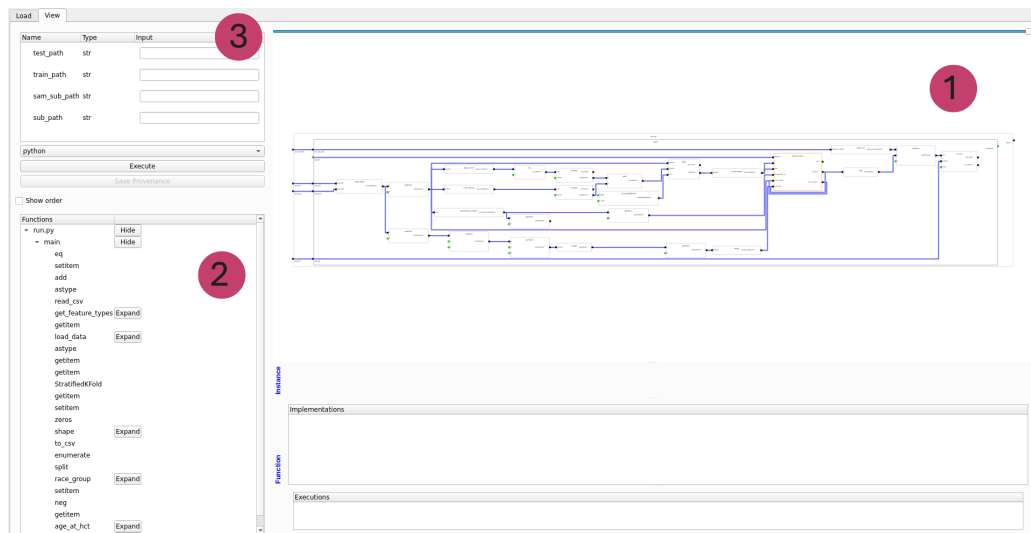


Figure 4: The FnO-Convert View Tab: Annotations are interactively visualized as connected function blocks and Composition mappings (Widget 1), with a collapsible tree overview of described functions and compositions (Widget 2). Functions can be executed with custom inputs, generating complete PROV-O provenance descriptions (Widget 3).

are automatically parsed upon loading, with their source code displayed in Widget 2. A dropdown in Widget 1 lets the user select a specific Function to inspect. The corresponding annotations (function, parameters, outputs, implementation, and mapping) are then displayed in Widget 3.

The View Tab (Figure 4) provides an interactive visualization of the generated annotations. Widget 1 displays each function as a block with parameters on the left and outputs on the right, with data flow between functions shown as blue lines representing Composition mappings. Widget 2 provides a tree overview of all described functions. Functions with a composition can be expanded or collapsed to reveal or hide their internal structure. Widget 3 demonstrates integration with retrospective provenance: users can provide input values and execute the described function, upon which the tool captures the resulting inputs and outputs, updates the visualization accordingly, and generates a PROV-O file recording the execution activities. This file can be downloaded via the save provenance button.

4. Conclusion

FnO-Convert automatically generates fine-grained, hierarchical prospective provenance from Python scripts without requiring developer annotations, spanning from top-level script structure down to individual function calls. By separating functional descriptions from their implementation environment, the generated descriptions are implementation-independent and interoperable. Dockerfile support demonstrates extensibility, showing how descriptions of heterogeneous artifacts can be meaningfully linked: RUN commands referencing Python scripts are internally described as invocations of those scripts with their specific inputs.

Future work will pursue four directions: enriching semantic descriptions by combining exhaustive extraction with human-provided annotations; extending support to additional languages and cross-language dataflow to cover multi-language pipelines; integrating with existing language tooling, such as the Language Server Protocol, to reduce reliance on custom parsing; and tighter integration with retrospective provenance systems.

Acknowledgments

This research was supported by the imec.icon project PACSOI (HBC.2023.0752), which was co-financed by imec and VLAIO and brings together the following partners: FAQIR Foundation, FAQIR Institute, MoveUP, Byteflies, AContrario, and Ghent University – IDLab.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-4 in order to: Grammar and spelling check. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

- [1] A. A. Shick, C. M. Webber, N. Kiarashi, J. P. Weinberg, A. Deoras, N. Petrick, A. Saha, M. C. Diamond, Transparency of artificial intelligence/machine learning-enabled medical devices, *npj Digital Medicine* (2024). doi:10.1038/s41746-023-00992-8.
- [2] S. B. Davidson, J. Freire, Provenance and scientific workflows: Challenges and opportunities, in: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1345–1350. doi:10.1145/1376616.1376772.
- [3] R. Souza, L. G. Azevedo, V. Lourenço, E. Soares, R. Thiago, R. Brandao, D. Civitarese, E. V. B. Nery, M. Moreno, P. Valduriez, M. Mattoso, R. Cerqueira, M. A. S. Netto, Provenance data in the machine learning lifecycle in computational science and engineering, in: *Workflows in Support of Large-Scale Science (WORKS)*, Workshop at SC, IEEE, 2019, pp. 1–10. doi:<https://doi.org/10.1109/WORKS49585.2019.00006>.
- [4] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, R. K. Bocinsky, Y. Cao, F. Chirigati, S. Dey, J. Freire, et al., YesWorkflow: A user-oriented, language-independent tool for recovering workflow information from scripts, *International Journal of Digital Curation* 10 (2015) 298–313. doi:10.2218/ijdc.v10i1.370.
- [5] D. Pina, L. Kunstmann, A. Chapman, D. de Oliveira, M. Mattoso, Dlprov: a suite of provenance services for deep learning workflow analyses, *PeerJ Computer Science* 11 (2025) e2985. URL: <https://peerj.com/articles/cs-2985>. doi:10.7717/peerj-cs.2985.
- [6] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, M. Weimer, Vamsa: Automated provenance tracking in data science scripts, in: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20)*, ACM, 2020, pp. 1542–1551. doi:10.1145/3394486.3403205.

- [7] K. Srinivas, I. Abdelaziz, J. Dolby, J. P. McCusker, Graph4Code: A machine interpretable knowledge graph for code, arXiv preprint arXiv:2002.09440 (2020). URL: <https://arxiv.org/abs/2002.09440>.
- [8] B. De Meester, T. Seymoens, A. Dimou, R. Verborgh, Implementation-independent function reuse, *Future Generation Computer Systems* 110 (2020) 946–959. doi:10.1016/j.future.2019.10.006.
- [9] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, J. Zhao, PROV-O: The PROV Ontology, W3C Recommendation, W3C, 2013. URL: <https://www.w3.org/TR/prov-o/>.
- [10] P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond assertions: Advanced specification and verification with JML and ESC/Java2, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2006, pp. 342–363.
- [11] N. Mitchell, Hoogole overview, *Monad.Reader* 12 (2008) 27–35.
- [12] I. García-Contreras, J. F. Morales, M. V. Hermenegildo, Semantic code browsing, *Theory and Practice of Logic Programming* 16 (2016) 721–737.
- [13] M. Atzeni, M. Atzori, CodeOntology: RDF-ization of source code, in: *The Semantic Web – ISWC 2017*, volume 10588, Springer, Cham, 2017, pp. 20–28.
- [14] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, K. Sycara, OWL-S: Semantic Markup for Web Services, Member Submission, World Wide Web Consortium (W3C), 2004. URL: <http://www.w3.org/Submission/OWL-S/>.
- [15] J. Van Herwegen, R. Taelman, S. Capadisli, R. Verborgh, Describing configurations of software experiments as linked data, in: D. Garijo, W. R. van Hage, T. Kauppinen, T. Kuhn, J. Zhao (Eds.), *Proceedings of the First Workshop on Enabling Open Semantic Science (SemSci)*, number 1931 in *CEUR Workshop Proceedings*, Aachen, 2017, pp. 23–30.
- [16] R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens, E. Della Valle, Representing dockerfiles in rdf, in: *Proceedings of the ISWC 2017 Posters, Demos and Industry Tracks*, volume 1963 of *CEUR Workshop Proceedings*, CEUR-WS, 2017. URL: <https://ceur-ws.org/Vol-1963/paper528.pdf>.