The Journal of Open Source Software

PipeOptz: A Python Library for Pipeline Optimization

Nicolas Pengov [1,2], **Théo Jaunet** [1,2], and **Romain Vuillemot** [1,2]

1 Ecole Centrale de Lyon, France **2** LIRIS CNRS UMR 5205, France

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Erik Whiting

Reviewers:

- @jqwang2373
- @gflofst

Submitted: 21 November 2025
Published: unpublished

Summary

PipeOptz is a Python library for the building, visualizing, and fine-tuning of processing pipelines. It enables users to define a series of operations as a DAG (Directed Acyclic Graph) which parameters can then be optimized to achieve a desired outcome. The library is designed to be suitable for a wide range of applications, and is particularly suited for image processing where workflows can be dense and often require parameter tuning.

Statement of need

In many scientific and engineering domains, complex data processing workflows are common. These workflows, i.e., pipelines, often consist of multiple steps, each with its own set of parameters and outputs. Finding the optimal set of parameters, and their individual influence, for a given task can be a tedious and time-consuming process which is often solved through manual trial and error. This is especially true in fields like image processing (Walt et al., 2014), where a sequence of filters and transformations is applied to an image, e.g., to find the best thresholding parameters. As opposed standard deep learning systems, pipelines, and their parameters have also the benefit to be more interpretable, through visualizations, and more easily reproducible.

Existing tools for pipeline management often fall into two categories: heavy-weight workflow orchestration frameworks (e.g., Airflow (*Apache Airflow*, n.d.), Prefect (*Prefect*, n.d.)) that are designed for large-scale data engineering tasks, or more specialized machine learning pipeline libraries (e.g., Scikit-learn pipelines (Pedregosa et al., 2011)) that are focused on linear sequences of operations. From our experience, we found a need for a lightweight, flexible, and Pythonic library that is suited for the easy creation, visualization, and optimization of, non-linear pipelines directly within a Python script.

`PipeOptz` addresses this need by providing an API for defining pipelines as Directed Acyclic Graphs (DAGs), with support for conditional branching and looping. In this graph, each node is a user-defined function in python, to ensure expressivity, and application to various domains. It integrates parameter optimization as a core feature, enabling users to define a search space for their pipeline's parameters and use various baseline optimization algorithms to find the best configuration.

State of the field

PipeOptz sits at the intersection of workflow orchestration, pipeline representation, and hyperparameter/black-box optimization. Workflow orchestrators such as Apache Airflow (*Apache Airflow*, n.d.) and Prefect (*Prefect*, n.d.) provide rich operational features (scheduling, monitoring, retries, deployments) and are well-suited for production batch workflows, but they are not designed as lightweight research libraries that expose the pipeline graph as a first-class object for iterative experimentation and optimization inside another tool. On the other end of the spectrum, hyperparameter optimization (HPO) frameworks and Bayesian optimization toolkits typically assume a user-written objective function and leave the internal structure of the computational pipeline implicit in the user's code, which limits explicit control-flow nodes, graph-level visualization, and step-wise traceability.

PipeOptz was created to support the needs of Descript, where we required (i) an expressive, multi-step pipeline with explicit control-flow (loops and conditionals), (ii) heterogeneous tunable parameters optimized against an application-specific loss function, and (iii) built-in graph visualization and execution traceability for rapid research iteration. In principle, part of this functionality could be implemented by extending an existing optimization toolkit with a custom loss, but the core requirement here is the combination of "pipeline-as-a-graph" modeling, control-flow nodes, and a clean separation between execution and optimization. Because these constraints cut across the fundamental abstractions of existing orchestration/HPO tools, we implemented a dedicated library designed as a reusable backend component for research workflows rather than an operational orchestrator.

To clarify our position with respect to closely related optimization libraries, Bayesian optimization frameworks such as BayesO (Kim & Choi, 2023) and pyGPGO (Jiménez & Ginebra, 2017) focus primarily on sample-efficient search strategies for expensive black-box objectives. In these systems the pipeline is usually encoded inside a single objective function, so the optimizer does not directly represent intermediate steps or control flow. PipeOptz keeps the optimization goal identical (minimize a user-defined loss), but makes the evaluation procedure explicit: the workflow is represented as a graph of nodes with dependencies and control-flow constructs, enabling node-level traceability and visualization while still treating the overall pipeline outcome as the quantity to optimize.

AutoML frameworks such as NiaAML (Pečnik & Fister, 2021) also address "pipeline + optimization", but they target the automated composition and tuning of machine-learning pipelines within a predefined space of ML components and objectives. PipeOptz is intentionally not ML-specific: it targets research workflows where the pipeline steps are arbitrary Python functions and the loss can encode domain-specific criteria (e.g., balancing geometric accuracy and the number of extracted targets), making it suitable as a backend for alternative approaches beyond conventional ML pipelines.

Finally, some optimization problems are best addressed by algebraic modeling and solver-based approaches. Linopy (Hofmann, 2023), for example, provides a modeling layer for linear and mixed-integer optimization with labeled n-dimensional variables and solver backends. PipeOptz is complementary: it targets workflows whose objective is evaluated by executing an end-to-end pipeline and cannot be naturally expressed as a linear/mixed-integer model.

Software Design

`PipeOptz` is designed to make research pipelines explicit and optimizable while keeping them lightweight and fully Python-native. The main design trade-off is to favor expressivity and traceability over an "objective-function-only" interface: instead of hiding the workflow inside a single function, `PipeOptz` represents it as a pipeline graph with explicit dependencies and control-flow nodes. This matters in research workflows where debugging, profiling, and iterating on multi-step processing chains is as important as finding good parameter values.

`PipeOptz` is built around the following core concepts:

- **`Node`**: The basic building block of a pipeline. A `Node` wraps a single Python function and its parameters. To support non-linear workflows beyond simple DAG composition, we provide dedicated control-flow nodes that embed sub-pipelines:
  - `NodeIf`: for conditional branching (if/else).
  - `NodeFor`: for 'for' loops.
  - `NodeWhile`: for 'while' loops.

- **`Pipeline`**: A `Pipeline` holds the entire workflow. Nodes are added to the pipeline with their dependencies, forming a DAG. The pipeline manages execution by following a topological order (Kahn, 1962). During execution, the library can cache node outputs and record per-node execution time, supporting fine-grained inspection and iterative refinement of complex pipelines.
- **`Parameter`**: A `Parameter` defines the type and search space for a value to be optimized. `PipeOptz` provides several types of parameters: `IntParameter`, `FloatParameter`, `ChoiceParameter`, `MultiChoiceParameter`, and `BoolParameter`.
- **`PipelineOptimizer`**: The optimization layer is separated from pipeline execution. It takes a pipeline, a set of parameters to optimize, and a user-defined loss function to minimize, and then evaluates candidate configurations by running the pipeline. `PipeOptz` provides several baseline optimization strategies :
  - Grid Search (GS),
  - Bayesian Optimization (BO) (Shahriari et al., 2016; Snoek et al., 2012),
  - Ant Colony Optimization (ACO) (Dorigo & Gambardella, 1997),
  - Simulated Annealing (SA) (Kirkpatrick et al., 1983),
  - Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995)
  - Genetic Algorithm (GA) (Holland, 1975).

The library also provides features for: - **Visualization**: Pipelines can be visualized as graphs using the `to_dot` and `to_image` methods, which generate Graphviz dot files and PNG images (Gansner & North, 2000). - **Serialization**: Pipelines can be saved to and loaded from JSON files using the `to_json` and `from_json` methods, enabling reuse and sharing beyond a single Python script (the `.dot` export is used for visualization and does not capture full pipeline semantics).

Research Impact Statement

PipeOptz is currently used as a workflow-level optimization backend in ongoing applied research prototypes where the target objective is not a standard machine-learning training loss, but an application-specific loss computed by executing a multi-step processing workflow. In these settings, practitioners need to iterate quickly over non-linear pipelines (including branching and loops) and tune heterogeneous parameters while keeping the workflow explicit, inspectable, and reproducible.

As this work is ongoing, we focus on credible near-term significance and reusability signals. PipeOptz is distributed as a Python package via PyPI, released under an OSI-approved license, and includes continuous integration, automated tests, and structured documentation with runnable examples. The repository provides executable examples demonstrating core capabilities (including control-flow pipelines and end-to-end optimization), and the runtime interface exposes node-level outputs and execution timing to support debugging and profiling of research workflows. These materials make PipeOptz reusable by other researchers who need to define, visualize, and optimize DAG-based pipelines in a lightweight, Pythonic way, especially in image-processing and related scientific workflows.

AI usage disclosure

We used generative-AI-assisted developer tools during software development and documentation writing, but not for drafting the JOSS manuscript.

Code: GitHub Copilot (Visual Studio Code extension) (last version used : 1.104.1) and Gemini Code Assist (Visual Studio Code extension) (last version used : 2.53) were used primarily for code completion and small refactoring suggestions during implementation. No large, unverified code blocks were accepted as-is; all AI-assisted edits were reviewed, tested, and integrated by the authors, who made the primary architectural and design decisions.

Documentation: Gemini Code Assist was used to accelerate writing of repetitive API documentation (docstrings, README sections). All generated text was manually reviewed and edited for correctness and consistency with the implemented behavior.

Manuscript: No generative AI tools were used to write `paper.md`.

Audience

`PipeOptz` is intended for researchers, data scientists, and engineers who need to build, visualize, and optimize data processing workflows in Python. It is particularly useful for those working in image processing, computer vision, and other scientific domains where pipeline-based workflows are common.

Example Usage

The following example demonstrates how to use `PipeOptz` to find the minimum of a simple function $f(x, y) = (x-3)^2 + (y+1)^2$ using Bayesian Optimization. The pipeline is constructed from multiple nodes to showcase the graph-based approach.

```
from pipeoptz import Pipeline, Node, FloatParameter, PipelineOptimizer

# Define the functions for the nodes
def squared_error(x, y):
    return (x - y)**2

def add(x, y):
    return x + y

# Create the pipeline
pipe = Pipeline("SimplePipeline")
pipe.add_node(Node("X", squared_error, fixed_params={"x": 0, "y": -3}))
pipe.add_node(Node("Y", squared_error, fixed_params={"x": -1, "y": 0}))
pipe.add_node(Node("Add", add), predecessors={"x": "X", "y": "Y"})

# The loss is the function's output, as we want to minimize it
def loss_func(result, _):
    return result

# Set up the optimizer with tunable parameters
optimizer = PipelineOptimizer(pipe, loss_function=loss_func)
optimizer.add_param(FloatParameter("X", "x", -5.0, 5.0))
optimizer.add_param(FloatParameter("Y", "y", -5.0, 5.0))

# Run the Bayesian Optimization
# We provide a dummy dataset ([{}] and [0]) as this example does not depend on external
best_params, loss_log = optimizer.optimize([{}], [0], method="BO", iterations=25, init_p

print("Best parameters found:", best_params)
print(f"Final loss: {loss_log[-1]:.4f}")
```

This script will search for the optimal values for `X.x` and `Y.y` that minimize the final output of the pipeline. The expected output will show the best parameters found, which should be close to `{'X.x': 3.0, 'Y.y': -1.0}`, and a final loss close to 0.

We can visualize the pipeline using Graphviz.

```
from PIL import Image
pipe.to_image("pipeline.png")
im = Image.open("pipeline.png")
im.show()
```

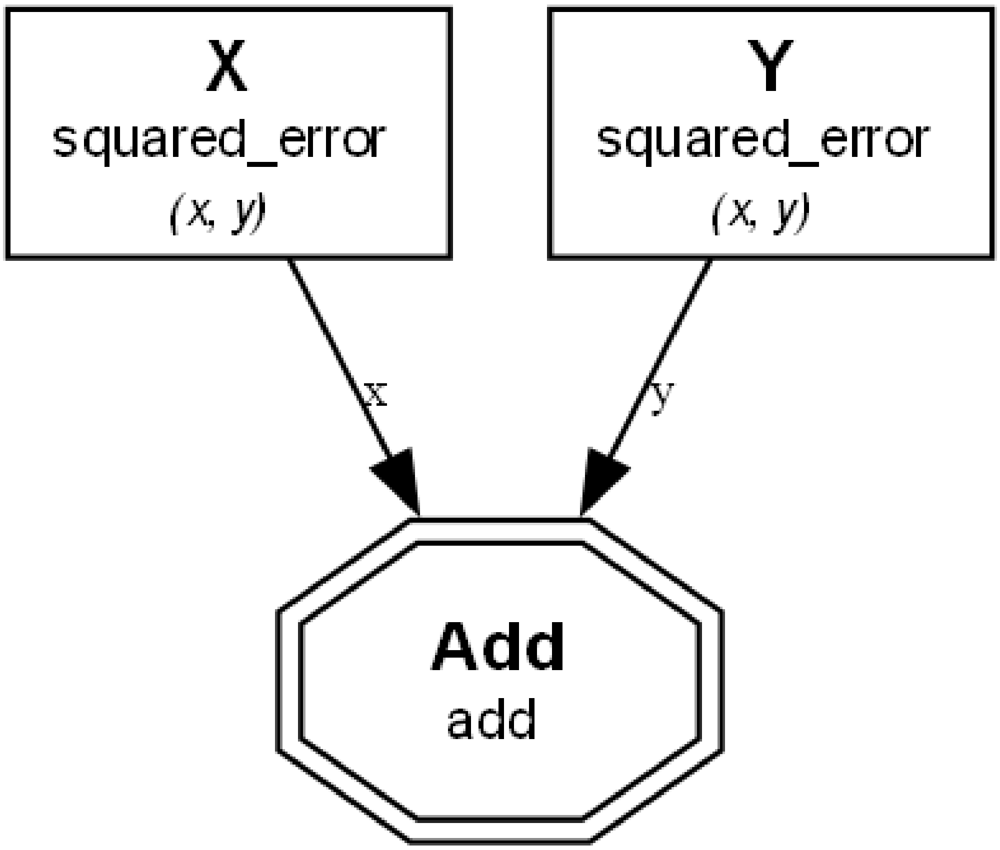

Figure 1: Visualization of the example pipeline.

Citations

`PipeOptz` complements lightweight helpers designed for algorithm evaluation (Küderle et al., 2023), and relies on NumPy (Harris et al., 2020), SciPy (Virtanen et al., 2020), and scikit-learn (Pedregosa et al., 2011) for numerical computing and Gaussian-process-based Bayesian optimization (Rasmussen & Williams, 2006). Its optimization engine builds on Bayesian Optimization techniques (Shahriari et al., 2016; Snoek et al., 2012) alongside the broader family of metaheuristics surveyed in (Tomar et al., 2023).

Acknowledgements

This research is partially funded by ANR, the French National Research Agency with the GLACIS project (grant ANR-21-CE33-0002).

References

Apache airflow. (n.d.). Retrieved December 21, 2025, from https://airflow.apache.org/

Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, *1*(1), 53–66. https://doi.org/10.1109/4235.585892

Gansner, E. R., & North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, *30*(11), 1203–1233. https://doi.org/10.1002/1097-024X(200009)30:11%3C1203::AID-SPE338%3E3.0.CO;2-N

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hofmann, F. (2023). Linopy: Linear optimization with n-dimensional labeled variables. *Journal of Open Source Software*, *8*(84), 4823. https://doi.org/10.21105/joss.04823

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press.

Jiménez, J., & Ginebra, J. (2017). pyGPGO: Bayesian optimization for python. *Journal of Open Source Software*, *2*(19), 431. https://doi.org/10.21105/joss.00431

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, *5*(11), 558–562. https://doi.org/10.1145/368996.369025

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Proceedings of the IEEE International Conference on Neural Networks*, 1942–1948. https://doi.org/10.1109/ICNN.1995.488968

Kim, J., & Choi, S. (2023). BayesO: A bayesian optimization framework in python. *Journal of Open Source Software*, *8*(90), 5320. https://doi.org/10.21105/joss.05320

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

Küderle, A., Richer, R., Sîmpetru, R. C., & Eskofier, B. M. (2023). Tpcp: Tiny pipelines for complex problems - a set of framework independent helpers for algorithms development and evaluation. *Journal of Open Source Software*, *8*(82), 4953. https://doi.org/10.21105/joss.04953

Pečnik, L., & Fister, I. (2021). NiaAML: AutoML framework based on stochastic population-based nature-inspired algorithms. *Journal of Open Source Software*, *6*(61), 2949. https://doi.org/10.21105/joss.02949

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830. http://jmlr.org/papers/v12/pedregosa11a.html

Prefect. (n.d.). Retrieved December 21, 2025, from https://www.prefect.io/

Rasmussen, C. E., & Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press. http://www.gaussianprocess.org/gpml/

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & Freitas, N. de. (2016). Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

Snoek, J., Larochelle, H., & Adams, R. P. (2012). *Practical bayesian optimization of machine learning algorithms*. https://arxiv.org/abs/1206.2944

Tomar, V., Bansal, M., & Singh, P. (2023). Metaheuristic algorithms for optimization: A brief review. *Engineering Proceedings*, *59*(1). https://doi.org/10.3390/engproc2023059238

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., Walt, S. J. van der, Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … Mulbregt, P. van. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Walt, S. van der, Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., Gouillart, E., Yu, T., & contributors, the scikit-image. (2014). Scikit-image: Image processing in Python. *PeerJ*, *2*, e453. https://doi.org/10.7717/peerj.453