Abdullah A Almaatooq, et. al. International Journal of Engineering Research and Applications
www.ijera.com
ISSN: 2248-9622, Vol. 16, Issue 9, September 2026, pp 30-36

RESEARCH ARTICLE OPEN ACCESS

The Role of Python in Artificial Intelligence and Machine Learning Development

Abdullah A Almaatooq[1], Alaa H Ridha[2]
Computer Department[1], Higher Institute of Telecommunication and Navigation[1]
Email: imatooq79@gmail.com
Computer Department[2], Higher Institute of Telecommunication and Navigation[2]
Email: hitn2014@gmail.com

ABSTRACT
Python has become a central language in artificial intelligence and machine learning because it pairs accessible syntax with a mature scientific-computing ecosystem. This paper examines how that pairing supports the full machine-learning lifecycle, from data preparation and exploratory analysis through model training, evaluation, deployment, and scaling. We review the roles of NumPy, pandas, SciPy, scikit-learn, TensorFlow, PyTorch, Jupyter, and related tools, and explain how Python can present a simple user-facing layer over optimized C, C++, and accelerator code. The discussion extends to computer vision, natural-language processing, generative AI, and scientific applications, and it weighs Python's advantages against limitations in interpreter overhead, dependency management, concurrency, reproducibility, and production maintenance. Our argument is that Python's importance has little to do with raw execution speed. Its greater contribution is the common development environment it provides for researchers, independent learners, data specialists, and software engineers. Continued work on compilers, distributed systems, packaging, and free-threaded execution should strengthen that role.

Keywords - artificial intelligence, deep learning, machine learning, Python, scientific computing

---

Date of Submission: 09-09-2026 Date of acceptance: 18-09-2026

I. INTRODUCTION

Artificial intelligence and machine learning have moved out of specialized research and into ordinary technology. Recommendation systems, language models, medical imaging tools, fraud detection, industrial monitoring, and autonomous systems all rest on software that learns patterns from data. As that base has widened, the programming environment used to build, test, and maintain such software has come to matter more. A language for modern AI work has to handle mathematical operations, data, rapid experimentation, hardware acceleration, collaboration, and deployment. Python is where most of these activities now meet.

Recent development data give a sense of the scale. GitHub reported roughly 2.6 million Python contributors in 2025, growth close to 49 percent year over year, and named the language the leading choice for AI and data-science projects [1]. The 2025 Stack Overflow Developer Survey recorded a seven-percentage-point rise in Python use and tied it to AI, data science, and back-end development [2]. The two use different methods, and neither should be read as a universal market share. What they do establish is that Python has outgrown classrooms and short scripts, and now runs through research, open-source development, and production software alike.

Popularity is not the same as technical importance. Python executed directly by the standard interpreter is slower than compiled code, and no large production system solves its data-quality, security, or maintenance problems through language simplicity. Python's real role is that of a coordination layer. It gives developers a concise way to describe an experiment while numerical kernels, tensor operations, and device-specific routines run in optimized native code, and it gives libraries a shared vocabulary for arrays, tabular data, estimators, model components, and visualization. The distance between an idea and a working experiment shrinks, and few users have to write low-level code to benefit.

We analyze that role at three levels. The first is the language itself: the features and interactive tools that suit iterative investigation. The second is the ecosystem supporting data science, classical machine learning, deep learning, and distributed workloads. The third is what happens once Python-based work leaves a notebook and becomes a maintained application, where the practical advantages and the limitations both appear. Our central argument is that Python became influential in AI by joining separate

communities and technical layers in one workable environment, so its value is measured less by execution speed than by how quickly and reliably people move through the complete development process.

II. FROM A GENERAL-PURPOSE LANGUAGE TO AN AI PLATFORM

A. Language characteristics and developer productivity

Python is a high-level, general-purpose language with dynamic typing, automatic memory management, modules, exceptions, object-oriented features, and support for several programming styles [3]. None of that makes it an AI language by definition. What it makes Python is flexible enough to express data-processing scripts, numerical experiments, web services, automation, and application logic without forcing a team to change languages at every stage. A researcher can test an algorithm in a short program, then reorganize the same ideas into functions, packages, tests, and services as the project grows.

Readable syntax matters in AI development for a mundane reason: model code is revised constantly. Feature definitions, preprocessing rules, loss functions, and evaluation procedures all change during an experiment, and a compact program makes it easier to inspect assumptions and compare alternatives. Readability also opens examples and teaching material to learners who understand statistics or domain science before they have much software-engineering experience. The entry barrier drops, though nothing about it removes the need to understand algorithms, data leakage, uncertainty, or experimental design.

Dynamic typing supports quick iteration and carries a cost. Functions and objects compose with little initial ceremony, but some interface errors surface only when a particular path executes. Larger Python projects therefore lean on type hints, static analyzers, validation libraries, automated tests, and clearly defined data schemas. Flexibility combined with engineering discipline is productive. Flexibility alone turns an easily written prototype into something no one wants to maintain.

B. Interactive and reproducible workflows

AI work rarely runs in a straight line from requirements to final code. Developers inspect distributions, correct missing values, plot samples, train a model, study the errors, and return to earlier decisions. IPython was built to improve exactly this interactive style of scientific computing, combining an enhanced shell with access to Python's wider numerical ecosystem [4]. Jupyter then developed the notebook into a document that holds executable code, output, equations, and explanation in one organized record [5]. Together these tools made experimentation visible and shareable instead of leaving it buried in disconnected scripts.

For exploration, teaching, demonstrations, and explaining a method, a notebook is hard to beat: it can show how a dataset was transformed and why a result came out as it did. It can equally damage reproducibility when cells run out of order, environments go unrecorded, or much of the pipeline stays manual. Mature projects usually pull reusable functions and tests out into modules while keeping notebooks for analysis and communication. Python's contribution is that both forms draw on the same libraries and data structures, so exploratory work can grow into a controlled codebase rather than being rewritten from scratch.

III. THE PYTHON AI AND MACHINE LEARNING ECOSYSTEM

The best explanation for Python's position is not any single feature of the language. It is the layered ecosystem around it, in which each layer addresses a different part of the workflow and the layers exchange data with little friction. Table 1 lists several components that have become common across education, research, and industry.

Table 1. Core Components of the Python AI/ML Ecosystem

Component	Primary contribution
NumPy	Arrays, vectorized numerical operations, and a shared data model
pandas	Cleaning, joining, reshaping, and analyzing structured data
scikit-learn	Classical algorithms, preprocessing, pipelines, and evaluation
TensorFlow / PyTorch	Deep learning, automatic differentiation, and accelerator access
Jupyter	Interactive experiments that combine code, results, and explanation

A. Numerical arrays, scientific routines, and data frames

NumPy supplies the multidimensional array model beneath most of scientific Python. Its array operations express calculations over entire collections of values instead of slow Python-level loops, and the project defines a shared interface that other libraries can accept or extend. Harris et al. describe NumPy as the foundation for array programming across a large scientific software ecosystem [6]. For machine learning, that foundation carries vectorized feature transformations, linear algebra, random sampling, and the movement of data between libraries.

SciPy builds on the same array model with algorithms for optimization, integration,

interpolation, signal processing, statistics, sparse matrices, and related technical tasks [7]. Those routines matter because an AI project is more than model fitting. Sensor data may need filters, an objective function may need numerical optimization, an evaluation may need statistical tests. Reaching all of it through consistent Python interfaces spares a team from assembling a separate toolchain for every supporting step.

pandas handles structured and time-oriented data through the Series and DataFrame abstractions. Its early design targeted the operations that real datasets demand: alignment, missing values, grouping, joining, reshaping, and labeled axes [8]. These are central to machine learning because raw data almost never arrive as the matrix an algorithm expects, and preparation often consumes more effort than model selection. Because pandas sits in the same environment as the modeling code, an analyst can trace a result back through the transformations that produced it.

B. Classical machine learning with scikit-learn

scikit-learn offers supervised and unsupervised algorithms behind a consistent estimator interface, covering preprocessing, feature selection, classification, regression, clustering, dimensionality reduction, model selection, and metrics. The project's stated design goals are usability, documentation, performance, and API consistency [9]. Because the fit, transform, and predict pattern holds across the library, comparing methods does not mean rewriting the complete pipeline for each one.

The library earns its place in practical learning and in medium-scale problems where a well-understood statistical or machine-learning method suits the task better than a deep neural network. Pipelines bind preprocessing to an estimator, which cuts the risk of treating training and test data differently, and cross-validation and search tools push toward systematic comparison. None of this guarantees a valid experiment. It does make good practice easier to implement and easier to review.

C. Deep learning frameworks and hardware acceleration

TensorFlow and PyTorch carried Python from traditional analysis into large-scale deep learning. TensorFlow represents computation as operations that can be placed across CPUs, GPUs, clusters, and specialized accelerators, with a system design aimed at research and production workloads at substantial scale [10]. PyTorch takes an imperative, Python-oriented approach in which tensor computation and automatic differentiation are expressed as ordinary program logic while still reaching hardware accelerators [11]. Both frameworks have changed considerably since, yet both illustrate the same pattern: Python defines the model and orchestrates the work while optimized runtimes perform the expensive calculations.

Automatic differentiation is much of why the arrangement works. The developer describes a forward computation and a loss function, and the framework constructs the derivative operations that gradient-based training requires. That removes a great deal of manual calculus and implementation work, and it makes models easier to modify, since gradients are recomputed when the architecture changes. Device management, batching, checkpointing, mixed precision, and distributed training stack on top, letting the same Python-level model run on progressively more capable hardware.

D. Natural-language processing and reusable models

Transformer models raised Python's importance again. The Transformers library exposes common Python interfaces to pretrained language models, tokenizers, training procedures, and evaluation utilities across the major deep-learning backends [12]. Reusable models changed the typical workflow. Rather than designing and training every architecture from the beginning, many teams now select a pretrained model, adapt it to the task, evaluate it on appropriate data, and integrate it into an application.

That convenience has a second face. A model downloads and runs in a few lines, but responsible use still turns on licensing, data provenance, bias, privacy, security, computational cost, and fitness for the domain. Python makes advanced models reachable; it does not make their outputs correct. The easier a system is to call, the more weight the surrounding tests and human review have to carry.

IV. PYTHON ACROSS THE MACHINE LEARNING LIFECYCLE

A. Data collection, exploration, and preparation

Python's first practical job is usually connecting data sources. Standard libraries and third-party packages read files, databases, web services, message queues, images, audio, and scientific formats, and the same program can validate fields, remove duplicates, normalize units, and produce diagnostic summaries. That breadth matters because model quality is bounded by the information and the errors already present in the data. No algorithm recovers a target that was recorded incorrectly or represents a group missing from the dataset.

Exploration is how such problems surface before training rather than after. Histograms expose extreme values, grouped summaries expose sampling

imbalance, and a careful look at the images exposes mislabeled ones. Python's array, table, plotting, and notebook tools all serve this work. A sound workflow records every transformation in code and saves the configuration that produced the training data. Editing a spreadsheet by hand is faster in the moment and makes the result nearly impossible to reproduce later.

B. Training, evaluation, and error analysis

During model development, Python holds feature engineering, algorithm configuration, training loops, metrics, and visualization in one place. Rapid comparison follows naturally, but speed cannot substitute for experimental control. Training, validation, and test sets need distinct purposes, and the metric should reflect what errors actually cost in the application. Accuracy misleads under class imbalance, and a strong average can conceal poor performance on a subgroup that matters. The libraries supply plenty of metrics; deciding which evidence answers the real question remains the developer's job.

Error analysis deserves the same attention. Confusion matrices, residual plots, per-class scores, and inspection of failed examples show where a model is weak. Since Python connects model outputs to general data-analysis tools, a team can group errors by source, time period, device, or user segment, which usually teaches more than another round of hyperparameter tuning. Systematic failure is a signal to improve the data or revise the problem definition, not to enlarge the network.

C. Deployment and distributed execution

A trained model is useful only once it can serve a real process. Python models may sit behind web APIs, run in scheduled batch pipelines, embed in data platforms, or be exported to portable formats, and the choice follows from latency, throughput, privacy, hardware, and maintenance requirements. For many systems a Python service is entirely adequate. For mobile, browser, or strict real-time environments, the trained model is usually converted and executed by a specialized runtime while Python stays the development and validation environment.

Distributed systems extend the same pattern to larger workloads. Ray, for instance, provides task and actor abstractions for distributed AI applications behind Python interfaces [13], and deep-learning frameworks coordinate multiple accelerators and machines on their own. Scaling becomes more accessible, and new failure modes arrive with it: communication, scheduling, state, and reproducibility. Moving from one computer to a cluster is an architectural change rather than a configuration switch.

Deployment also means monitoring. Data distributions drift, upstream fields vanish, and user behavior moves away from the conditions represented during training. A Python-based monitoring process can compare current inputs and outcomes against training baselines, provided each alert is tied to a decision someone will actually make. Recording model versions, feature definitions, dependency versions, and evaluation results is what later makes a failure investigable and a return to a known model possible.

V. MAJOR AREAS OF APPLICATION

A. Computer vision

Computer-vision systems use Python for image loading, augmentation, annotation processing, model training, and evaluation. Deep-learning frameworks contribute convolutional layers, vision transformers, pretrained networks, and GPU execution, while general scientific libraries handle the filtering, geometry, statistics, and visualization around the model. The combination serves tasks from document recognition and manufacturing inspection to medical-image research. In high-stakes settings, evaluation has to run on data that represent the target population and the actual acquisition conditions, not on whichever benchmark is convenient.

B. Natural-language processing and generative AI

Natural-language processing leans on Python for text cleaning, tokenization, dataset construction, model adaptation, retrieval, and evaluation, and the availability of pretrained transformer models has shortened the path from an idea to a demonstration. Python also covers the components a generative-AI application needs around the model: vector search clients, document parsers, prompt management, evaluation scripts, and service APIs. Producing an output is rarely the hard part. Establishing that the output is accurate, safe, relevant, and consistent enough for its intended use is.

C. Science, engineering, and business analysis

Many AI projects sit on top of domain calculations. A scientific application may need differential equations or signal processing; a business application may need time-based aggregation and cost analysis. Python's scientific ecosystem lets those computations live beside the learning algorithms, so a model can be judged by the physical or economic consequences of its predictions and not only by a statistical score. That link is especially valuable in applied projects, where it keeps machine learning framed as one component of a larger problem rather than an isolated set of algorithms.

VI. ADVANTAGES FOR DEVELOPERS AND ORGANIZATIONS

Three advantages reinforce one another. The first is development speed: concise syntax and

interactive tools let a team test a data transformation or a modeling idea quickly. The second is ecosystem coverage, with mature libraries for numerical computing, data management, visualization, conventional machine learning, deep learning, natural-language processing, distributed execution, web services, and automation. The third is communication, since researchers, analysts, and software engineers can exchange code and results without translating every experiment into a different language.

Open-source development compounds all three. Widely used projects attract contributions, documentation, examples, and integrations from institutions around the world, and a common array model with familiar API conventions makes it straightforward for a new package to work with existing tools. Educational adoption then supplies more developers who already know the environment. The cycle feeds itself: libraries attract users, users contribute improvements and examples, and the expanded ecosystem attracts further projects.

Interoperability counts as much as any native feature. NumPy, TensorFlow, PyTorch, and similar systems push performance-critical operations into compiled implementations; Python extensions call C and C++; tensor frameworks drive CUDA and other device runtimes. This hybrid architecture avoids a straight choice between productivity and performance, with Python expressing the high-level program while specialized code handles dense numerical work. The boundary is not free, and a badly written Python loop is still slow, but well-designed libraries keep most of the low-level complexity behind stable interfaces.

Knowledge portability is a further advantage, and an underrated one. A framework's exact API will change, yet core skills in arrays, data frames, validation, model evaluation, and Python packaging transfer across projects. Training costs fall and multidisciplinary teams form more easily. It also explains why Python stays useful when the final inference runtime is written in another language: the experiments, the reference implementation, the tests, and the data analysis still share one foundation.

VII. LIMITATIONS AND ENGINEERING TRADE-OFFS

A. Runtime performance and concurrency

The standard CPython interpreter runs ordinary Python code more slowly than optimized native code. Dynamic object handling, interpreter dispatch, and memory overhead become significant in tight loops and very large object collections. Vectorized NumPy operations and tensor frameworks push that work into compiled kernels, and just-in-time compilation offers another route; Numba, for example, uses LLVM to compile selected numerical Python functions into machine code [14]. The gains can be substantial. Realizing them requires knowing which parts of a program are actually suitable for compilation or vectorization.

Concurrency has long been constrained by the Global Interpreter Lock in the usual CPython build, above all for CPU-bound threads that manipulate Python objects. Multiprocessing, native extensions, asynchronous I/O, and distributed systems each provide a way around it, and each adds complexity. Python 3.13 introduced an optional free-threaded build, and Python 3.14 moved that build to officially supported status under PEP 779 [15]. The direction is important, though existing extensions and applications still need testing and redesign before parallel threads translate into faster programs.

B. Dependencies, environments, and reproducibility

An ecosystem this large brings dependency problems with it. AI packages may depend on specific versions of Python, numerical libraries, GPU drivers, and device toolkits, and a program that runs on one computer can fail on another when those layers differ. Virtual environments, lock files, containers, continuous integration, and recorded hardware information all improve reproducibility, and all of them belong to the project from the start rather than to a cleanup phase before deployment.

Notebooks add a risk of their own, because the visible order of cells need not match the execution history. A result can rest on hidden state created by a cell that ran earlier and has since been edited. Restarting the kernel and running the notebook from beginning to end is a crude test that catches most of this. Important transformations belong in version-controlled modules, and datasets should be identified by stable versions or checksums, so the work stays reviewable when its author is unavailable.

C. From successful model to maintainable system

Machine-learning code is only a small part of a production system. Data pipelines, configuration, monitoring, access control, feedback loops, and user interfaces often hold more logic than the model does. Sculley et al. describe how ML systems accumulate technical debt through changing data dependencies, hidden feedback, and complex interactions between components [16]. Python's flexibility helps a team wire these pieces together, and it makes informal dependencies just as easy to create. Clear module boundaries, schemas, tests, code review, and named ownership are what keep the system tractable.

Security belongs in the same discussion. Loading an untrusted serialized object or model package can execute harmful code, and any dependency can introduce vulnerabilities. Credentials

do not belong in notebooks, downloaded artifacts should come from trusted sources, and deployment permissions should follow the principle of least privilege. Outputs need validation too: a language model can produce plausible but false statements, and a classifier can be confidently wrong on unfamiliar data. Safeguards have to be designed around these failure modes instead of assuming that a high test score covers every situation.

VIII. ALTERNATIVES AND INTEROPERABILITY

Python is not the best language for every component. C and C++ remain essential for kernels, runtimes, embedded systems, and strict latency requirements. R offers a deep statistical environment and holds strong positions in many research fields. Java and C# are common in enterprise systems, while JavaScript and TypeScript are the natural choices for browser interfaces. Julia was designed to combine a high-level dynamic language with strong numerical performance and multiple dispatch [17]. Workload, team, deployment target, and existing systems should decide the choice, not popularity.

Modern AI systems are multilingual in practice. A Python training pipeline may call C++ and GPU kernels, export a model to a portable runtime, serve predictions from a container, and hand results to a TypeScript application. Python holds up in that setting because it can act as the development and integration language even when it is not the final execution layer. Good architecture makes these boundaries explicit, measures what data transfer costs, and preserves tests that compare the production runtime against the Python reference implementation.

IX. FUTURE DIRECTIONS

Python's future in AI depends on improvements above and below the language itself. Compiler projects, vectorized libraries, graph capture, and specialized runtimes keep lowering the performance cost of high-level code. Distributed frameworks make clusters usable without exposing every systems detail. Free-threaded CPython may improve CPU parallelism for suitable applications once libraries become compatible. Packaging standards and environment tools, meanwhile, still have to answer the difficulty of reproducing complex hardware and software combinations.

AI-assisted programming will reshape the ecosystem as well. Code-generation tools explain APIs and produce initial implementations, which can bring new users into Python quickly, though the output still requires testing, security review, and an understanding of the underlying algorithm. The sheer volume of public Python code and documentation makes the language convenient for such tools. Convenience spreads weak patterns as readily as strong ones, so education should stress reading, debugging, evaluation, and source verification rather than treating generated code as reliable by default.

The most durable direction is deeper interoperability. Array standards, portable model formats, shared compiler infrastructure, and stable extension interfaces let Python programs reach new hardware and languages without abandoning established workflows. As long as those interfaces stay open and well documented, Python can go on serving as a meeting point between algorithm research, domain expertise, and production engineering.

X. CONCLUSION

Python became central to artificial intelligence and machine learning because it supports far more than model implementation. It connects data preparation, scientific computation, visualization, conventional algorithms, deep learning, evaluation, deployment, and monitoring inside one coherent environment. NumPy, pandas, SciPy, scikit-learn, TensorFlow, PyTorch, and Jupyter give the language capabilities that syntax alone could never explain. Optimized native kernels and accelerators supply the performance; Python supplies the interface through which people develop and connect the system.

The limitations are real. Direct interpreter performance, concurrency, dependency conflicts, notebook state, security, and production technical debt all demand deliberate engineering. Python should not be chosen by reflex, and it should not be mistaken for the complete runtime of an AI application. Its strongest role is as a productive coordination layer resting on specialized implementations and clear interfaces.

For independent learners and new practitioners, Python opens an accessible route into AI, and learning the language is only the beginning. Reliable work also demands statistics, algorithmic understanding, attention to data quality, reproducible experiments, software testing, and ethical judgment. Paired with those practices, Python's ecosystem remains one of the most effective platforms for turning machine-learning ideas into systems that can be examined, improved, and used responsibly.

REFERENCES

[1] GitHub Staff, 'Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1,' GitHub Blog, Oct. 28, 2025. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/. Accessed: Aug. 30, 2026.

[2] Stack Overflow, '2025 Developer Survey: Technology,' 2025. [Online]. Available:

https://survey.stackoverflow.co/2025/technology. Accessed: Aug. 30, 2026.

[3] Python Software Foundation, 'The Python Language Reference,' Python 3.14 Documentation, 2026. [Online]. Available: https://docs.python.org/3/reference/index.html. Accessed: Aug. 30, 2026.

[4] F. Perez and B. E. Granger, 'IPython: A System for Interactive Scientific Computing,' Computing in Science & Engineering, vol. 9, no. 3, pp. 21-29, May-June 2007, doi: 10.1109/MCSE.2007.53.

[5] 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, doi: 10.3233/978-1-61499-649-1-87.

[6] C. R. Harris et al., 'Array programming with NumPy,' Nature, vol. 585, pp. 357-362, 2020, doi: 10.1038/s41586-020-2649-2.

[7] 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.

[8] W. McKinney, 'Data structures for statistical computing in Python,' in Proceedings of the 9th Python in Science Conference, 2010, pp. 56-61, doi: 10.25080/Majora-92bf1922-00a.

[9] F. Pedregosa et al., 'Scikit-learn: Machine Learning in Python,' Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011. [Online]. Available: https://jmlr.org/papers/v12/pedregosa11a.html.

[10] M. Abadi et al., 'TensorFlow: A System for Large-Scale Machine Learning,' in 12th USENIX Symposium on Operating Systems Design and Implementation, 2016, pp. 265-283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[11] A. Paszke et al., 'PyTorch: An Imperative Style, High-Performance Deep Learning Library,' in Advances in Neural Information Processing Systems 32, 2019, pp. 8024-8035. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[12] 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, 2020, pp. 38-45, doi: 10.18653/v1/2020.emnlp-demos.6.

[13] P. Moritz et al., 'Ray: A Distributed Framework for Emerging AI Applications,' in 13th USENIX Symposium on Operating Systems Design and Implementation, 2018, pp. 561-577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/moritz.

[14] S. K. Lam, A. Pitrou, and S. Seibert, 'Numba: A LLVM-based Python JIT Compiler,' in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015, article 7, pp. 1-6, doi: 10.1145/2833157.2833162.

[15] T. Wouters, M. Page, and S. Gross, 'PEP 779 - Criteria for supported status for free-threaded Python,' Python Enhancement Proposals, Mar. 13, 2025. [Online]. Available: https://peps.python.org/pep-0779/. Accessed: Aug. 30, 2026.

[16] D. Sculley et al., 'Hidden Technical Debt in Machine Learning Systems,' in Advances in Neural Information Processing Systems 28, 2015, pp. 2503-2511. [Online]. Available: https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.

[17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, 'Julia: A Fresh Approach to Numerical Computing,' SIAM Review, vol. 59, no. 1, pp. 65-98, 2017, doi: 10.1137/141000671.