Tesseract Core: Universal, autodiff-native software components for Simulation Intelligence

Dion Häfner [1¶] **and Alexander Lavin** [1]

1 Pasteur Labs, Brooklyn, NY, USA ¶ Corresponding author

DOI: 10.21105/joss.08385

Software

- Review
- Repository
- Archive

Editor: Chris Vernon

Reviewers:

- @gartavanis
- @aslan-ng

Submitted: 14 May 2025
Published: 22 July 2025

Summary

Tesseracts are universal software components for scientific computing, simulation, and machine learning (ML), summarized as "simulation intelligence" (SI) (Lavin et al., 2021). Specifically, Tesseracts enable and expedite the transition from experimental, research-grade software to production environments. This includes native support for automatic differentiation between heterogeneous software artifacts in distributed and cloud contexts, which enables end-to-end differentiable programming, hybrid ML and simulation systems, and more for SI at scale.

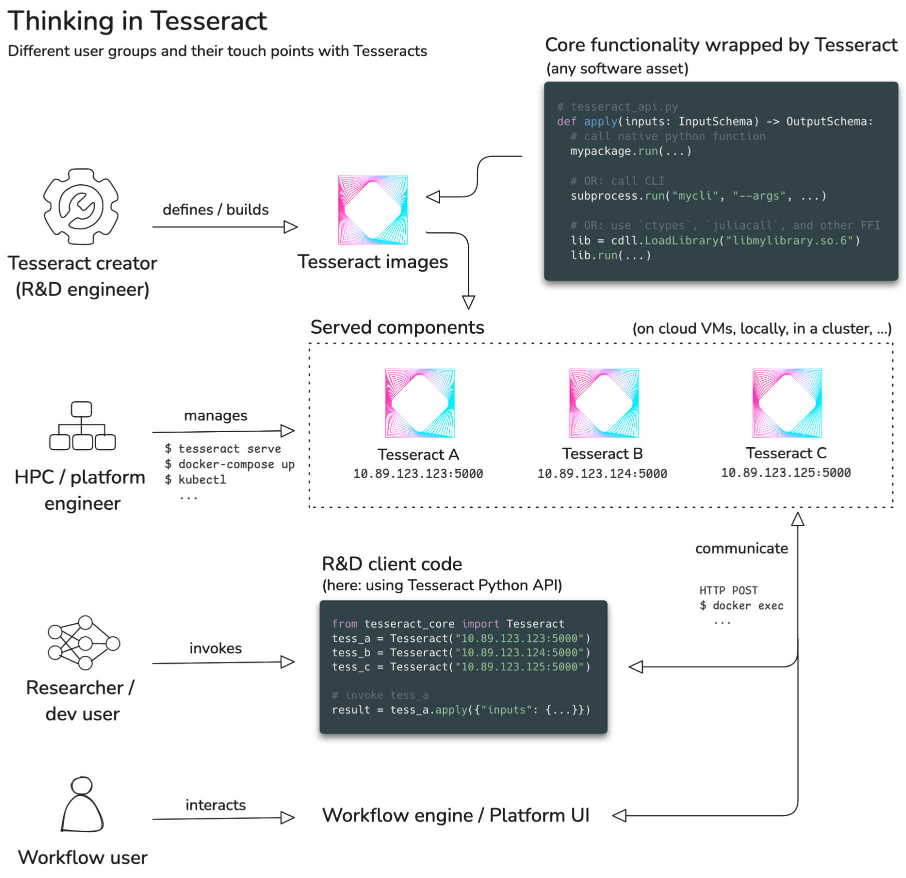
Tesseract Core is a Python library that serves as the reference implementation for defining, containerizing, executing, and deploying Tesseract components. It provides user entry points to wrap existing software artifacts such as Python functions, Julia code, C++ libraries, or remote services into Tesseract components. By unambiguously defining allowed inputs and outputs of each Tesseract via Pydantic models (Colvin et al., 2025), Tesseract Core enables external data validation and auto-generation of machine-readable API schemas. This allows users to explore the capabilities of Tesseract components without interacting with code, and enables workflow engines to compose them into larger, self-validating pipelines that are, by the virtue of implementing Tesseracts, end-to-end differentiable. Tesseract components are built to be deployed in distributed contexts, including support for containerization and remote procedure calls (RPC) via network.

Statement of need

The last few years have seen a rapid increase in the use of machine learning and data-driven methods within scientific computing and industrial simulation. However, when moving beyond the research lab and into real-world engineering workflows, the gap between (GPU-accelerated, autodiff-native, high-level, bare-bones) research code and (CPU-bound, non-differentiable, low-level, feature-rich) production code is often too large to bridge. Similar issues arise when integrating components of different software stacks (e.g., Python, Julia, C++), and when deploying components in heterogeneous or distributed environments (e.g., cloud vs. on-prem, high-performance computing (HPC), and supercomputers).

These software bottlenecks translate to significant untapped potential of differentiable programming in particular and simulation intelligence in general—for example, when exploring the merits of hybrid AI-simulator systems in differentiable physics contexts (Avila Belbute-Peres et al., 2018; Kochkov et al., 2024; Um et al., 2020), or when using machine learning to augment existing simulation codes (Freitas et al., 2024; Shankar et al., 2025).

Tesseract Core remediates these bottlenecks and elevates integration possibilities by acting as glue between several user groups (Figure 1):



```python
# tesseract_api.py
def apply(inputs: InputSchema) -> OutputSchema:
  # call native python function
  mypackage.run(...)

  # OR: call CLI
  subprocess.run("mycli", "--args", ...)

  # OR: use `ctypes`, `juliacall`, and other FFI
  lib = cdll.LoadLibrary("libmylibrary.so.6")
  lib.run(...)
```

```
$ tesseract serve
$ docker-compose up
$ kubectl
  ...
```

```python
from tesseract_core import Tesseract
tess_a = Tesseract("10.89.123.123:5000")
tess_b = Tesseract("10.89.123.124:5000")
tess_c = Tesseract("10.89.123.125:5000")

# invoke tess_a
result = tess_a.apply({"inputs": {...}})
```

Figure 1: Creating, deploying, and using Tesseracts through the lens of various user groups.

1. **Researchers** building multi-component systems and experiment pipelines: Tesseract Core provides consistent interfaces to interact with any Tesseract component in the same way. This allows users to discover and download existing Tesseract containers, lowering the bar for researchers to experiment with many different components when developing *systems* rather than single *operations*.
2. **R&D software engineers** building research tools and packaging them: Tesseracts are defined via a Python-based interface with minimal configuration. Users specify input/output schemas for each Tesseract, enabling transparent I/O validation, automatic differentiation, and remote execution behind a unified interface. Tesseract Core provides the tools to validate and build Docker containers from Tesseract definitions.
3. **Platform and HPC engineers** building SI workloads at scale: Tesseracts can be executed in any environment that supports the Tesseract runtime. This allows them to be embedded into virtually any orchestration framework and executed on bare metal, in the cloud, or on compute clusters. Tesseracts expose their input/output schemas according to the machine-readable OpenAPI format, facilitating automated integration into external workflows engines.

This is a markedly different scope than what is found in existing software solutions, which typically focus on a single aspect of the problem space—such as containerization (e.g., Docker, Singularity), remote procedure calls (e.g., gRPC), automatic differentiation (e.g., JAX, PyTorch), and container orchestration (e.g., Kubernetes, Docker Compose)—or are geared towards specific use cases, like Gymnasium (Brockman et al., 2016) for reinforcement learning, or UM-Bridge (Seelinger et al., 2023) for probabilistic programming and HPC. Tesseract Core aims to unify the most valuable parts of these aspects into a single, coherent ecosystem that makes it easy to use and deploy SI methods across different environments.

Concretely, Tesseract Core addresses many of the fundamental issues that arise when building real-world SI systems, including but not limited to:

- **Code sharing** – How do I make my research-grade simulation code available to other users who are not familiar with the codebase?
- **Reproducibility** – How can I ensure that my simulation code is executed in a consistent and reproducible manner, regardless of the environment?
- **Streamlined experimentation** – How can I experiment with different 3rd party simulators, differentiable meshing routines, or ML models without having to install all their dependencies or study their documentation in depth?
- **Dependency management** – How do I resolve conflicts between different software or hardware requirements of components working together in a pipeline?
- **Remote execution** – How do I execute my SciML software on a cloud VM and query it from my local machine?
- **Explicit interfaces** – How do I discover at a glance which parameters of a software branded as "differentiable simulator" are differentiable and which are not?
- **Distributed differentiable programming** – How do I propagate gradients end-to-end through complex pipelines mixing torch, JAX, and Julia code?

Specific end-to-end examples that leverage the Tesseract ecosystem to address these issues in research and engineering settings are highlighted and continuously published in the Tesseract community showcase.

Software design

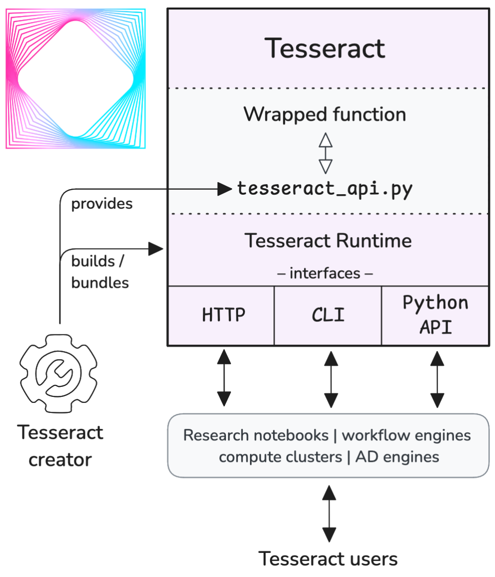
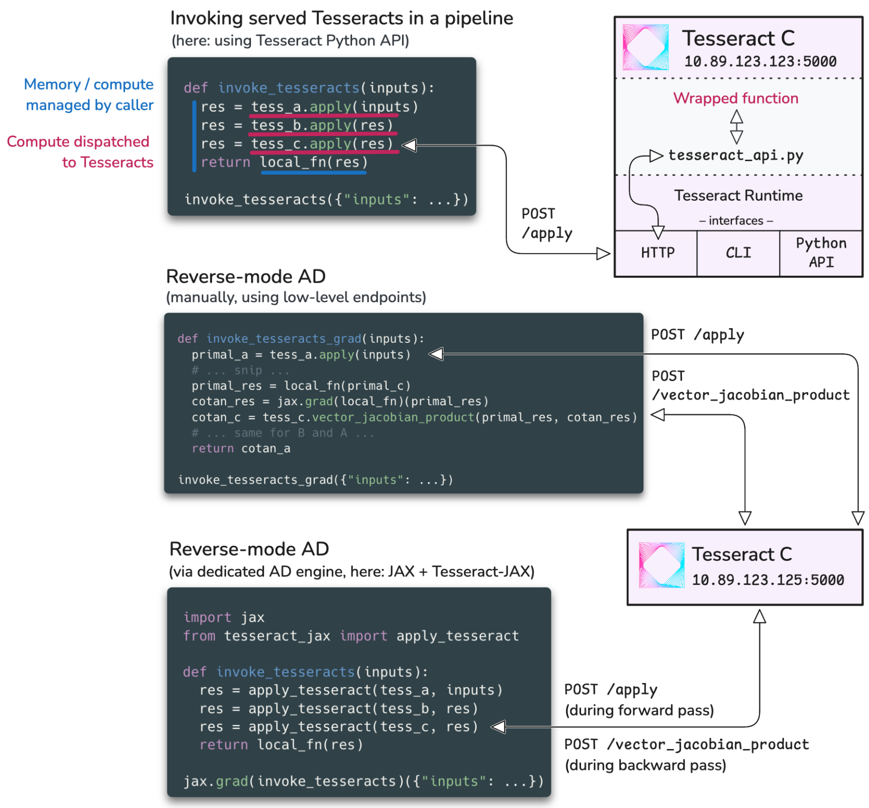


Figure 2: The make-up of a Tesseract highlighting its external and internal interfaces.

A Tesseract is any object that is served behind the Tesseract runtime, which ships with Tesseract Core and bundles a command-line interface (CLI), Python API, and REST API. Each of these external interfaces maps to the same internal code path, which ultimately invokes a `tesseract_api.py` Python module, provided by the Tesseract creator (Figure 2).

The structure of `tesseract_api.py` (and thus the Tesseract interface itself) is centered around a functional programming style without internal state. Specifically, this means that each Tesseract is assumed to wrap a single operation (`apply`) that takes a set of (arbitrarily nested) inputs and produces a set of outputs. All other Tesseract endpoints like `jacobian`, `abstract_eval`, or `vector_jacobian_product` represent transformations of the `apply` function. This is strongly inspired by JAX primitives (Bradbury et al., 2018), and enables the efficient use of automatic differentiation (AD) techniques such as reverse-mode AD and forward-mode AD (Ma et al., 2021), invoked manually or through existing AD engines (Figure 3).



Invoking served Tesseracts in a pipeline
(here: using Tesseract Python API)
Memory / compute managed by caller
Compute dispatched to Tesseracts
POST /apply
Tesseract C
10.89.123.123:5000
Wrapped function
tesseract_api.py
Tesseract Runtime
– interfaces –
HTTP
CLI
Python API
Reverse-mode AD
(manually, using low-level endpoints)
POST /apply
POST /vector_jacobian_product
Tesseract C
10.89.123.125:5000
Reverse-mode AD
(via dedicated AD engine, here: JAX + Tesseract-JAX)
POST /apply
(during forward pass)
POST /vector_jacobian_product
(during backward pass)

```
def invoke_tesseracts(inputs):
  res = tess_a.apply(inputs)
  res = tess_b.apply(res)
  res = tess_c.apply(res)
  return local_fn(res)

invoke_tesseracts({"inputs": ...})
```

```
def invoke_tesseracts_grad(inputs):
  primal_a = tess_a.apply(inputs)
  # ... snip ...
  primal_res = local_fn(primal_c)
  cotan_res = jax.grad(local_fn)(primal_res)
  cotan_c = tess_c.vector_jacobian_product(primal_res, cotan_res)
  # ... same for B and A ...
  return cotan_a

invoke_tesseracts_grad({"inputs": ...})
```

```
import jax
from tesseract_jax import apply_tesseract

def invoke_tesseracts(inputs):
  res = apply_tesseract(tess_a, inputs)
  res = apply_tesseract(tess_b, res)
  res = apply_tesseract(tess_c, res)
  return local_fn(res)

jax.grad(invoke_tesseracts)({"inputs": ...})
```

Figure 3: Example data flow through a Tesseract-driven compute pipeline, both during forward application and reverse-mode AD. When using a Tesseract-aware AD engine, implementation details of Tesseract endpoints (such as `apply` vs. `vector_jacobian_product`) are hidden from the user.

These methods and the myriad scenarios in which they can be applied continue to be real-world validated in diverse scientific contexts, engineering physics applications, and industrial simulation environments. Progress in these areas and more can be found in the Tesseract open-source community, at https://si-tesseract.discourse.group.

References

Avila Belbute-Peres, F. de, Smith, K., Allen, K., Tenenbaum, J., & Kolter, J. Z. (2018). End-to-end differentiable physics for learning and control. *Advances in Neural Information Processing Systems*, *31*.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/jax-ml/jax

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba,

W. (2016). OpenAI gym. *arXiv Preprint arXiv:1606.01540*. https://doi.org/10.48550/arXiv.1606.01540

Colvin, S., Jolibois, E., Ramezani, H., Garcia Badaracco, A., Dorsey, T., Montague, D., Matveenko, S., Trylesinski, M., Runkle, S., Hewitt, D., Hall, A., & Plot, V. (2025). *Pydantic* (Version v2.11.4). https://github.com/pydantic/pydantic

Freitas, A., Um, K., Desbrun, M., Buzzicotti, M., & Biferale, L. (2024). Solver-in-the-loop approach to turbulence closure. *arXiv Preprint arXiv:2411.13194*. https://doi.org/10.48550/arXiv.2411.13194

Kochkov, D., Yuval, J., Langmore, I., Norgaard, P., Smith, J., Mooers, G., Klöwer, M., Lottes, J., Rasp, S., Düben, P., & others. (2024). Neural general circulation models for weather and climate. *Nature*, *632*(8027), 1060–1066. https://doi.org/10.1038/s41586-024-07744-y

Lavin, A., Krakauer, D., Zenil, H., Gottschlich, J., Mattson, T., Brehmer, J., Anandkumar, A., Choudry, S., Rocki, K., Baydin, A. G., & others. (2021). Simulation intelligence: Towards a new generation of scientific methods. *arXiv Preprint arXiv:2112.03235*. https://doi.org/10.48550/arXiv.2112.03235

Ma, Y., Dixit, V., Innes, M. J., Guo, X., & Rackauckas, C. (2021). A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–9. https://doi.org/10.1109/hpec49654.2021.9622796

Seelinger, L., Cheng-Seelinger, V., Davis, A., Parno, M., & Reinarz, A. (2023). UM-bridge: Uncertainty quantification and modeling bridge. *Journal of Open Source Software*, *8*(83). https://doi.org/10.21105/joss.04748

Shankar, V., Chakraborty, D., Viswanathan, V., & Maulik, R. (2025). Differentiable turbulence: Closure as a partial differential equation constrained optimization. *Physical Review Fluids*, *10*(2), 024605. https://doi.org/10.1103/PhysRevFluids.10.024605

Um, K., Brand, R., Fei, Y. R., Holl, P., & Thuerey, N. (2020). Solver-in-the-loop: Learning from differentiable physics to interact with iterative PDE-solvers. *Advances in Neural Information Processing Systems*, *33*, 6111–6122.