Instrumenting Lightweight Modular Machine Learning Training and Inference in Parallel Solvers

Ayman Yousef[1], Corey Wetterer-Nelson[2], Mengjiao Han[3], Victor Mateevitsi[3], Joseph Insley[3], Silvio Rizzi[3], Janet Knowles[3], Michael E. Papka[3,4], and Amanda Randles[1]

[1] Duke University, Durham, NC 27709, USA
[2] Atomic Industries, Denver, CO 80014, USA
[3] Argonne National Laboratory, Argonne, IL 60439, USA
[4] University of Illinois Chicago 1200 West Harrison Street, Chicago, IL 60607, USA

Abstract. Recent advances in exascale computing have increased the resolution and fidelity of large-scale simulations, while rapid progress in deep learning has accelerated efforts to couple machine learning with physics-based solvers. We present a lightweight, modular in situ coupling methodology that embeds machine learning training and inference directly into simulation workflows using the ParaView and Catalyst APIs. The approach provides C++/Python interoperability via a solver-side data adaptor that packages simulation state into Conduit Nodes and a Catalyst-driven Python "bridge script" that converts solver fields into NumPy/PyTorch representations with minimal intrusion into the solver code. We describe the design and instrumentation required to integrate the framework and demonstrate it within a proxy (mini-app) of the HARVEY vascular flow solver. To illustrate practical usage, we implement both in situ training and in situ inference of a point-cloud autoencoder running concurrently with the solver. We report scalability and overhead characteristics and show that the approach enables distributed online ML workflows without language unification or major solver refactoring.

Keywords: In Situ · Machine Learning · HPC · Fluid Dynamics

1 Introduction

Machine learning (ML) is increasingly integrated into physics-based computational modeling, often as surrogate models or as augmentations to existing solvers [20]. These methods support digital-twin proxies for computationally intensive simulations [17] and solver adaptation through trainable recurrent or corrective models [8]. At the same time, the growing imbalance between compute capability and I/O throughput has motivated in situ workflows that analyze, compress, or visualize data during execution rather than post hoc by way of writing full simulation states to disk [2–4]. In situ execution is particularly well matched to ML training and inference because it can replace expensive solver-to-disk data

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

movement with ephemeral, runtime data access, while also making model outputs available immediately for monitoring or feedback. The overall structure of the proposed coupling methodology is shown in Figure 1.

Despite this potential, coupling simulation codes and ML models across heterogeneous programming languages remains challenging. Many approaches require unifying languages across solver and model [6, 16], substantial solver refactoring [5, 15], or building bespoke libraries [8, 19]. In addition, many toolchains target only inference or only training, limiting practical adoption due to an under-instrumentation of these complementary workflows. Finally, existing studies often provide limited guidance on the code changes required to orchestrate a production solver, making it difficult for developers to reproduce or extend prior approaches. This paper addresses these gaps with the following contributions:

- **A lightweight, Catalyst-based framework** for embedding both in situ ML training and inference into parallel solvers via a solver-side adaptor and Python scripts.
- **An integration in a HARVEY mini-app**, providing a reproducible proxy that demonstrates solver-side instrumentation and data movement patterns.
- **Two end-to-end use cases** that exercise both training and inference using a distributed point-cloud autoencoder running concurrently with the solver.

2 Related Work

The coupling of ML models and scientific solvers often falls into two broad categories: those that homogenize the language across all codes in an experimental setup and those that employ intermediary APIs to handle data conversion between the two ends. The inception and development of online machine learning in the open-source computational fluid dynamics model OpenFOAM [11] is exemplary of the former category. The work of Maulik et al. [16, 15] implements in situ machine learning in the OpenFOAM solver through two discrete approaches. Their preliminary work utilized TensorFlow's C API to load saved model "graphs" at runtime to enable concurrent inference [16]. These graphs were created before the simulation and stored offline. They further evolved their implementation, embedding a Python interpreter within the OpenFOAM solver to establish the PythonFOAM framework [15]. This approach facilitated direct data packaging in a Pythonic format to be used by models defined in Python through the Python/C API. PythonFOAM requires compiling with Python's C libraries and building functionality for data conversion from C++ to Python. This dependency, combined with the specificity of the use case, prompted concerns over invasiveness and generalizability. The work of Sun et al. [19] addressed some of the issues described by utilizing direct invocation of functions in Python through the use of the CFFI Python module and dynamic link libraries. The ability to use a dynamic link library broadens the applicability of their proposed framework. However, the lack of a model training workflow, the use of a less

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

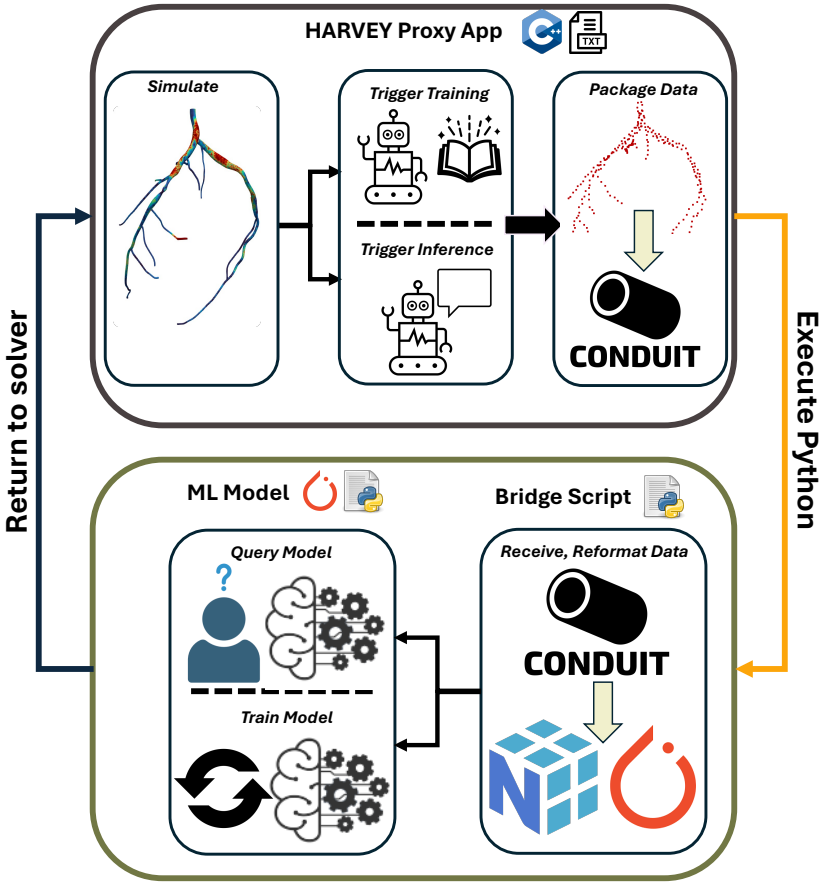

Fig. 1. In situ ML pipeline integrated into the HARVEY mini-app. Solver fields are packaged into a Conduit Node by a solver-side `CatalystAdaptor` and passed to Catalyst at runtime. Catalyst executes a Python "bridge script" (via ParaView) that converts Conduit views into NumPy/PyTorch tensors and invokes model-side training/inference routines. This separation keeps solver modifications minimal while enabling flexible Python-based ML workflows.

established Python module, and the absence of demonstrable GPU usage limit the utility of the described in situ ML scheme.

Balin et al. [5], and Kurz et al.'s [12] implementations of in situ machine learning center around the SmartSim application, representing the latter of the two coupling strategies. Balin et al. [5] deployed the SmartSim and SmartRedis applications with the PHASTA fluid dynamics solver, performing supervised model training and inference in situ. Kurz et al. [12] similarly utilized the SmartSim application to facilitate the recurrent training of an agent model tasked with optimizing eddy-viscosity coefficients of the FLEXI application. Although

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

the SmartSim application is more mature than the presented orchestration, the need to build both SmartSim and SmartRedis APIs and compile the solver with SmartSim increases the cost of deployment. There is also the issue of having to redesign the method for solver deployment. SmartSim-based applications require the use of a Python-based driver script to handle both solver and model execution. The additional requirement of reformatting major aspects of the workflow to conform to API specifications (ex., simulation and machine learning codes must be instantiated via SmartSim's "Model" wrappers to interface with the API-specific "Experiment" object) further burdens the developer. What's missing are applications that require less extensive code refactoring and lower the barrier to integrating in situ ML workflows in scientific solvers. In contrast, our approach leverages a widely deployed in situ ecosystem to provide a minimally invasive path to both training and inference without requiring solver language unification or a new workflow driver.

3 Design Considerations

We designed the framework around three guiding requirements: (i) minimal solver intrusion, (ii) support for both training and inference in situ, and (iii) compatibility with existing HPC software ecosystems. Several coupling strategies were considered. Direct language unification (e.g., embedding Python or using C++ ML APIs) simplifies interoperability but introduces significant solver refactoring and dependency constraints. Middleware-based approaches (e.g., SmartSim) provide robust orchestration but require restructuring solver execution and introducing additional runtime services. We instead adopt a Catalyst/ParaView-based design [1, 4], leveraging its widespread availability on leadership-class systems and its ability to decouple solver instrumentation from model execution. Conduit enables zero-copy data exchange through memory views, avoiding unnecessary duplication and reducing data movement overhead. This design prioritizes portability and low integration cost while still enabling distributed training and inference workflows within the solver execution context.

4 Implementation

We detail the three distinct components of the experimentation workflow: the proposed in situ machine learning framework, the solver in which it's integrated (termed solver-side), and the machine learning model invoked (termed model-side), with an overview of the framework in Figure 1. On the solver-side, we present the minimal adaptations made to the HARVEY solver to enable in situ workflows. Similarly, we detail the small alterations to model scripting needed to accommodate the in situ environment. We present our approach of linking the distinct C++ and Python domains into a cohesive system through the employment of the Catalyst and ParaView APIs. The Pythonic capabilities of the two APIs, in combination with the lightweight nature of Catalyst, its widespread

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

availability on leadership-class systems, and amenability to middleman APIs, make the presented framework highly integrable within existing codebases.

4.1 Framework Overview

HARVEY: One of the first considerations made during the design process was the choice of computational solver. The fluid-structure-interaction software package HARVEY was chosen due to its scalability on leadership-class systems, portability, and prior amenability to in situ analysis methodologies [18, 14, 24]. HARVEY is a lattice Boltzmann method (LBM) based solver adept at modeling the blood flow of patient-derived geometries of extensive, complex vasculature [21]. As a result, simulations often scale to the size of billions of fluid points to accurately recover patient-specific hemodynamics. We specifically employ a mini-app of the HARVEY solver due to the private nature of the codebase and the desire to provide a reproducible example. While chosen as the representative, we believe the framework to be deployable with like-solvers. Outside of compatibility with Catalyst/Conduit data formatting specifications and correct builds, there is nothing exclusive about the HARVEY solver that enables it to perform in situ machine learning with the methodology presented.

Catalyst and ParaView: The crux of data handling and connectivity between the two code domains is the Catalyst and ParaView packages. Catalyst is a software package that provides functionality for interrogating scientific data producers and creating visuals in situ [4]. Previously, Catalyst was a library packaged with the ParaView visualization software [1, 4]. As of ParaView v5.9, Catalyst is available as a standalone API. This "stub" is a bare-bones specification that provides the structure needed at compile time for instrumenting the necessary data adaptors. The ability to circumvent solver compilations with more complex packages like ParaView, instead allowing the use of dynamically opened libraries, ensures that the implementation is lightweight and portable. Compilation against the stub enables the use of Catalyst's three core functions: *initialize, execute, and finalize*. Catalyst handles the packaging of data from C to Python and interfaces with ParaView, whilst ParaView handles the execution of Python code. ParaView builds its own Python interpreter based on a provided base installation, providing the mechanism through which we are able to deploy machine learning models at runtime. Facility-maintained ParaView installations can be found across a multitude of leadership-class computing systems (e.g., Tuolumne, Frontier, Aurora), emphasizing the accessibility of the proposed framework. It is important to note that the inclusion of ParaView in the loop presents the prospect of using other synergistic functions. Native visualization capabilities and robust back-end compatibility are relevant advantages that can be co-opted to augment training workflows.

PyTorch: PyTorch is a seminal Python deep learning library that enables graph-based model programming [10]. The choice of PyTorch for model scripting

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

over other similar APIs came down to its availability on leadership-class systems, popularity amongst developers, and native support for distributed workflows. To enable the use of PyTorch functionality, we utilize a separate virtual environment containing the necessary libraries. The ability to use virtual environment libraries in tandem with ParaView is a considerably new feature of the software. Given that the Python installation used for the build of ParaView matches that of the standalone virtual environment and the appropriate library paths are defined, the virtual environment can be employed by ParaView seamlessly without the need for rebuilds or package reinstallations.

4.2 Solver-Side Adaptations

To utilize the proposed framework for Python code execution, there are a small number of non-invasive adjustments to the solver of interest required in order to interface with Catalyst. These adjustments include the creation and termination of a Catalyst instance at the beginning and the end of the solver, respectively, and the definition of execution functions. Within HARVEY, these Catalyst functions are defined within a class named `CatalystAdaptor`. `CatalystAdaptor` defines functions that directly mirror and invoke Catalyst's three core functions, labeled Initialize, Execute, and Finalize, to reflect their Catalyst counterparts. The Initialize function sets up the paths to relevant libraries (Catalyst, ParaView) and the script it is to run, while the Finalize function handles resource de-allocation. From a developer's perspective, the instrumentation cost is intentionally small. In practice, solver-side changes are limited to adding a `CatalystAdaptor` wrapper (initialize/execute/finalize) and packaging selected fields into a Conduit Node at user-defined timesteps. Model-side changes are limited to defining training/inference entry points callable from a middleman script and initializing distributed state variables once per run.

`CatalystAdaptor`'s Execute routine encapsulates both the data packaging and the call to Catalyst's *execute* function, as highlighted in Figure 2. Within this routine, HARVEY fluid data is converted to match the structure specified by the Conduit "Node" class [9]. Conduit provides a standard to describe computational simulation meshes through the definition of a mesh, a topology, and scalar fields in a hierarchical data structure similar to that of JSON or XML files (Figure 2A and 2B). Values can either be copied or passed as pointers, a zero-copy approach that serves to further underscore the lightweight nature of the proposed approach and seamlessly enable bidirectional data movement needed for routines like model inference. Conduit is commonplace within the in situ community, utilized within seminal APIs like Ascent [13], SOMA [22], and Catalyst. The widespread adoption of Conduit as a means of describing solver data, along with exhaustive examples outlined within online documentation, speaks to the convenience of our instrumentation. Once a Node object is populated, it is passed as the lone input argument to Catalyst's *execute* function (Figure 2C).

The script directly run by Catalyst is a middleman Python file termed the "bridge script", as illustrated in Figure 1. The script acts as the intermediary

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

between the simulation solver and the machine learning model, tasked with handling the data conversion from the Conduit Node object to fit general Python routines. The path to the bridge script is passed as a command-line argument and set within CatalystAdaptor's Initialize function. We access the passed Node data within the bridge script through the "catalyst-params" field, which returns a reference to the arguments passed in the Execute call in the form of a Conduit Node (Figure 2D). Conduit's Python specification defines Node objects as containers for C++ Node pointers. Conduit provides support for converting references to NumPy arrays through appropriate channel indexing (Figure 2E).

In the current implementation, the bridge script also handles the integration of the virtual environment by augmenting the list of Python library paths with those of the virtual environment. Additionally, the invocation of the ML model occurs within the bridge script. As highlighted in Figure 2F, we define our model Python script independently and load it in as a module within the bridge script. This compartmentalization allows the use of already defined model scripts, adding to the framework's versatility. Once data is passed and reformatted, training and inference functions defined by the model script can be called within the bridge script due to Python's modular programming.

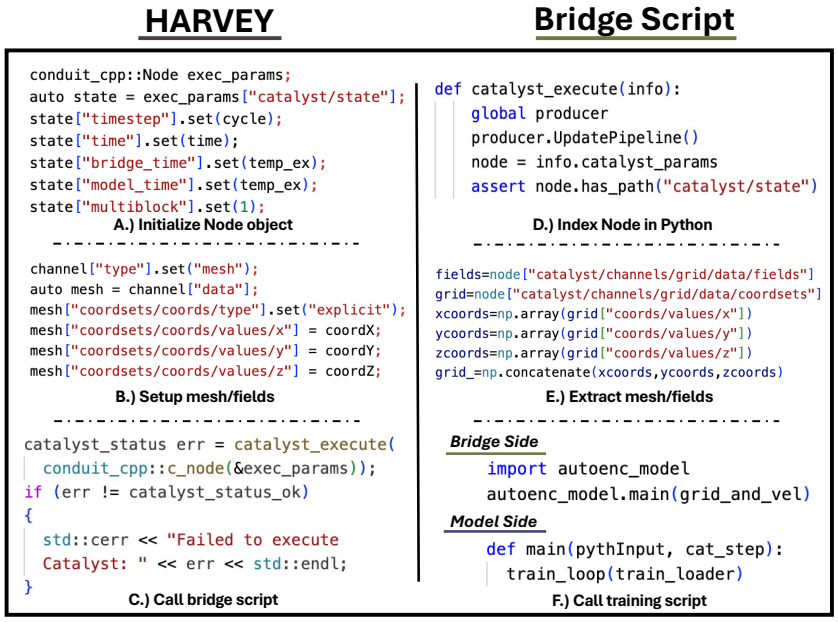

Fig. 2. Representative instrumentation code. (A–C) Solver-side `CatalystAdaptor` initialization and execution: simulation fields are packaged into a Conduit Node and passed to Catalyst. (D–F) Bridge-script logic: Catalyst parameters are accessed, Conduit arrays are converted to NumPy arrays, and model-side routines are invoked.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

4.3 Model-Side Adaptations

Model Instantiation and Distribution: The pervasiveness of model training and inference throughout the lifetime of the solver requires proper variable initialization. Model objects are initialized as global variables within model scripts across all studies, avoiding the need for additional data transfers and recurrent setup costs that characterize repeated setup. Alongside a model instance, we initialize rank, world size, and communication backend variables that are persistent throughout the lifetime of the simulation. Distributed training is supported using the PyTorch `DistributedDataParallel` (DDP) wrapper. MPI is used to obtain rank and world-size information (via the `mpi4py` library), while gradient synchronization is performed using PyTorch distributed collectives through the NVIDIA Collective Communications Library (NCCL) software layer. Each process calculates mini-gradients that are averaged across and passed back to all participating ranks to mimic the process of training a single model instance. We initialize an NCCL communication backend to enable the calculation and accumulation of gradients on GPUs. For training and inference, separate model instances are initialized per participating rank. Execution of the solver and ML routines occurs in a synchronized timestep-driven manner. This avoids race conditions by ensuring that data exchange occurs only at well-defined execution boundaries. Within the ML workflow, PyTorch DistributedDataParallel enforces synchronization of gradients across ranks using collective communication primitives, ensuring consistency across processes.

Training paradigms: In situ execution enables multiple training schedules that differ from conventional offline workflows. A direct analogue to offline training is to accumulate samples during the run and train once near the end of the simulation; however, this can require substantial memory to store the dataset. This is especially true in time-series models that require substantial length scales of data. An alternative is streaming or dynamic training, where each invocation both adds new solver samples and performs one or more optimization steps on the evolving dataset [5]. This approach can reduce peak storage requirements and supports responsive updates that are useful for feedback-driven settings (e.g., reinforcement learning or adaptive surrogates) [12]. Because these trade-offs are often underspecified in prior work, we evaluate both a "train-at-end" schedule and a dynamic, periodically trained schedule within our framework.

Data Bidirectionality: A distinction between the training and inference routines is the data exchange between the model and solver. For inference, data has to be relayed back to the calling model in order to update states. Within our pipeline, this bidirectionality between model and solver is seamless thanks to the zero-copy attribute of Conduit. Predictions are returned to the bridge script as NumPy arrays (or tensors converted to NumPy), and the bridge script writes these values back to Node fields that already reference solver-allocated buffers. This setup enables in-place updates on the solver side when the corresponding Conduit arrays are constructed as views of solver memory.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

5 Evaluation

To exhibit the presented framework and evaluate model performance, we carried out weak scaling studies to characterize scalability, profiled the overhead cost incurred as a result of in situ training, and disseminated two separate, distinct use cases of in situ machine learning focused on the training and query of a point cloud autoencoder. All testing occurred on the Polaris supercomputer (Argonne National Laboratory, NVIDIA). Across all experiments, we maintain a ratio of 1:1 between ranks and GPUs, with a total of 4 MPI ranks per node. Example model/bridge scripts and solver code can be found at the cited Github repository [23]. The primary sources of memory overhead arise from dataset buffering and model state replication across ranks.

5.1 Profiling model training stability and overhead

Given the novelty and prototypical nature of our pipeline, we began experimentation by first verifying the parallel training implementation. We constructed a point-cloud autoencoder model based on the work of [25] and trained it on fluid data generated by HARVEY to serve as a lossy compressor. An autoencoder model was specifically chosen for three reasons: its effectiveness at gauging the pipeline's capability of driving both model training and inference, the well-established utility of autoencoders as a means of lossy compression in LBM solvers [7], and its prevalence within in situ machine learning applications [5, 15]. We simulated steady velocity flow through the inlet of an idealized cylindrical domain across 64 ranks, performing in situ training throughout the simulation's lifetime. Training was invoked every 10 steps within the simulation's 100 timestep lifetime, with the fluid domain passed as training data (stacking with each subsequent training call) for the autoencoder. Training was performed for 10 epochs at each training call, with the training loss per epoch tracked across the simulation lifetime. Figure 3 showcases the convergence of training across the lifetime of training, with ranks convening at a similar loss value at $\sim$50 epochs into training. This convergence verifies that the communication scheme provided by PyTorch's DDP wrapper is correct and that the communication setup, as specified with the model training script, works as expected.

To understand the additional overhead attributable to the proposed pipeline and to assess its scalability, we performed a set of weak scaling runs on up to 64 nodes (equivalent to 256 MPI ranks). We kept the fluid domain size roughly consistent, aiming for $\sim$1 million fluid points per MPI rank. To fairly assess the nature of the pipeline, we set up a trivial training task, reducing the number of epochs per call to 1. We profiled the main portions of the framework to capture the overhead contribution of each aspect of the in situ pipeline, culminating in the results showcased in Figure 4. The blue bar encapsulates all HARVEY's typical simulation kernels, the green bars are representative of the model training routine, and the purple bars are indicative of the setup and invocation of model training associated with the CatalystAdaptor and bridge codes, respectively. Across the board, model training and the accompanying synchronization were

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

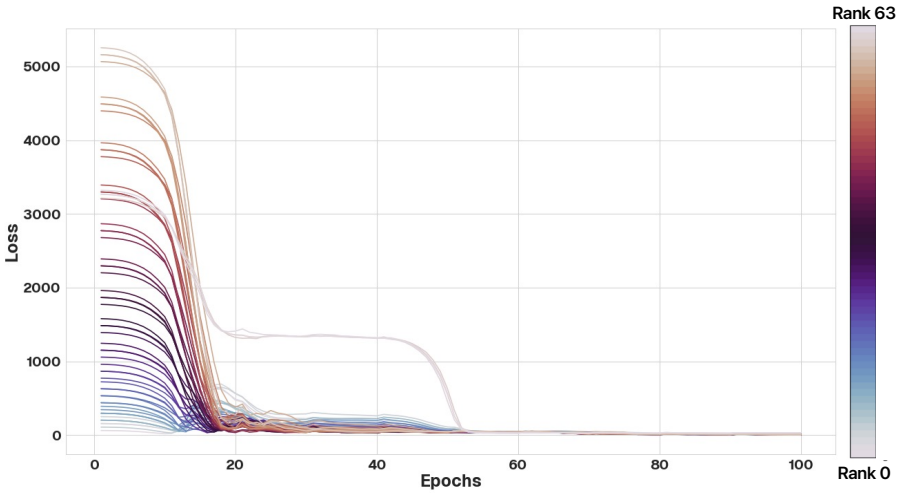

Fig. 3. Distributed in situ training stability. Training loss per epoch across ranks for an autoencoder trained concurrently with the solver. Convergence to a consistent loss across ranks indicates correct DDP synchronization and well-posed parallel training.

the most burdensome portion of the in situ training pipeline. The dominance of model synchronization is evident across all node counts past the single-node run. The data transit from solver to model proved significantly less costly across all simulation rank counts, with a max. cost of ∼2.9 seconds. Initial results suggest that a majority of the runtime is concentrated on the first timestep, which involves the setup of the global variables and initializing kernels, with repeated calls to train inducing a much lower cost as a result.

5.2 In situ autoencoder training on HARVEY fluid data

With confirmation of the training workflow, we sought to evaluate our methodology's ability to enable two distinct online machine learning workflows: online training/offline inference and offline training/online inference. For the former composition, we set out to investigate the two aforementioned training schemes in an online setting. The first mimics the generic offline training setup, training all at once with a stable dataset at the end of the simulation. The second explores training iteratively, with a dynamically populated training dataset that grows throughout the lifetime of the simulation. HARVEY simulation composition matched that of the verification experiment, albeit over a longer period of time to allow the fluid flow to fully develop. We simulated for a total of 1001 timesteps, with the first 800 timesteps dedicated to model training. This strategy reflects a potential use case in complex patient-specific fluid modeling. If effective, fluid data pre-convergence can be used to train a deep learning model on the task of accurately compressing domains at time points further along in

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

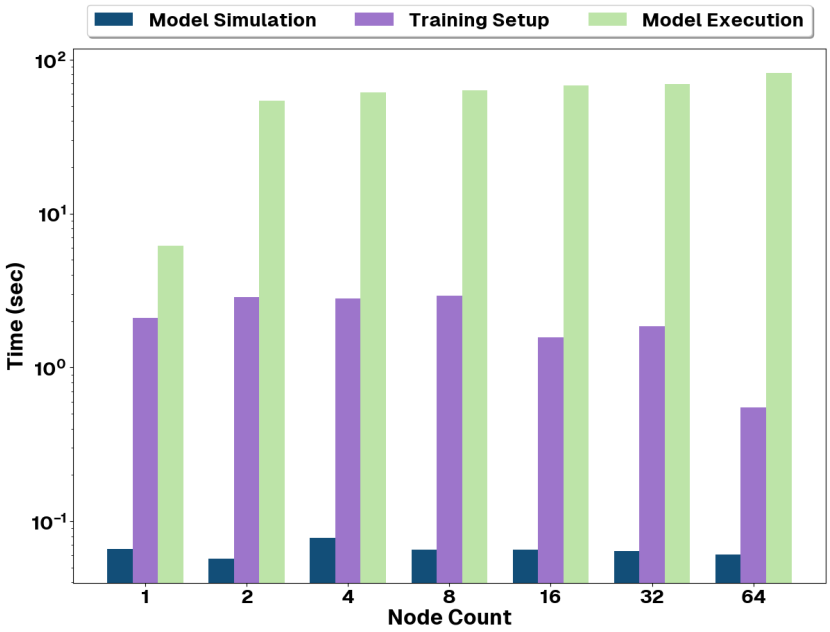

Fig. 4. Weak-scaling overhead attribution for in situ training. Total time is decomposed into solver compute (HARVEY kernels), model training (and synchronization), and framework overhead (data adaptation and bridge-script invocation). Training dominates at scale while solver-to-model linking remains comparatively small.

the simulation. Fluid domain data was collected every 10 timesteps to populate a custom PyTorch dataset crafted to handle a continuous stream of data. The two in situ training paradigms explored required slightly different directives within the bridge script. For training executed in step with the data generator, the bridge script invoked training every 100 timesteps, with each call looping through 10 epochs for a total of 80 training epochs throughout the simulation lifetime. In contrast, training was invoked once at the last timestep when investigating the traditional training scheme. We launched in situ training on a scaling set of MPI ranks and inputs (on up to 32 ranks) for both considered paradigms. We present the training results for the 4 MPI rank training case in Figure 5. Although the model is not overly performant, we compared training loss with a model trained with the same data offline, represented by the dotted yellow line, to ensure training was being performed correctly. Conventional training in situ outpaced iterative training, quickly reaching convergence within the first 10 epochs of training. Iterative training converged around the same loss value later in the lifetime of the model, proving the validity of the training scheme. Both in situ training schemes matched the performance of their offline counterpart, further verifying the fidelity of the solver-model coupling.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

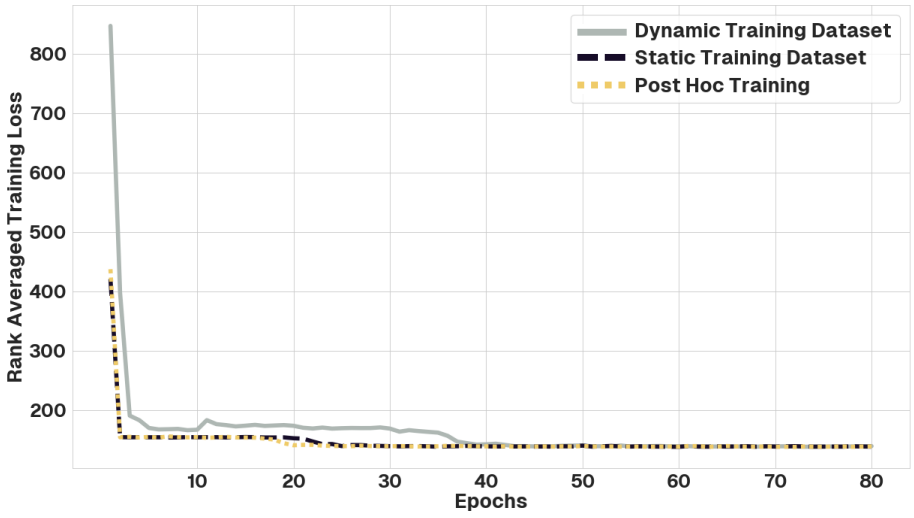

Fig. 5. Model loss curves generated across three separate training paradigms: in situ training with a static dataset, in situ training with an iteratively updated ("dynamic") dataset, and offline training. Loss curves were calculated based on the loss per epoch averaged across all 4 MPI ranks. Both in situ training schemes matched their offline training counterpart, with convergence around 10 epochs and 40 epochs, respectively.

5.3 Compressing HARVEY data through online model inference

To evaluate the second of the two focused online machine learning workflows, we initialized and trained an autoencoder model offline for testing of online inference functionality. We evaluated the ability of the trained model to encode the solver-generated fluid grid and accompanying velocity field through inference, with the same split between training and prediction timesteps as detailed in the online training study. Mimicking the simulation composition used for in situ training evaluation, we simulated flow within a cylinder for a total of 1001 timesteps. The latter 200 timesteps were reserved for offline inference to mirror the previous training-based experiment. We performed model inference every 10 timesteps, encoding the spatial domain and velocity field and saving the latent vector on disk. After the fact, we investigated the accuracy of model encoding by decoding the saved latent vectors to recover the original solver-based grid representation. An example reconstructed fluid grid for a single rank simulation is shown in Figure 6, visualized alongside the simulation-based ground truth. Although testing was performed in varying simulation sizes and rank sets, the serial experiment best illustrated the potential cost savings that lossy encoding can produce. The simulation domain was composed of a vector of size (21504, 4), while the latent encoded vector was composed of a vector of size (128, 1), leading to an effective compression rate of 672x. To evaluate reconstructive accuracy, we calculated both the relative Frobenius error (as defined in [5]) and the mean

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

squared error (MSE). The reconstruction, illustrated in Figure 6, had a relative Frobenius error of 0.0311 and an MSE of 2.441 and 8.667e-13 for the coordinates and velocity magnitude fields, respectively.

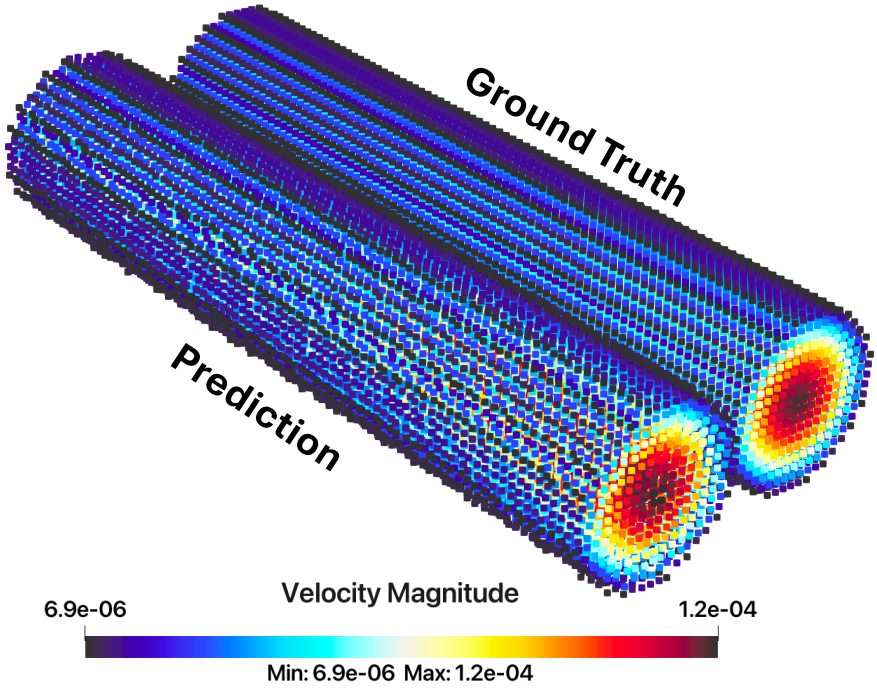

Fig. 6. Representative reconstruction from in situ inference. A pretrained autoencoder encodes solver state during the simulations and outputs latent vectors on disk. Post hoc decoding recovers the fluid-point coordinates and velocity magnitude, with the reconstructed field preserving the inlet-flow structure with minimal reconstruction error.

6 Conclusion

We presented a lightweight, modular framework for distributed in situ ML training and inference built on the Catalyst and ParaView software. By integrating the framework into the HARVEY mini-app, we demonstrated end-to-end coupling between a parallel fluid solver and Python-based ML workflows, including both in situ training and inference of a point-cloud autoencoder during solver execution. The design—compiling only against the Catalyst stub, using runtime-loaded libraries, and isolating ML logic in Python bridge/model scripts—provides a portable and minimally invasive path for adding ML capabilities to existing solvers without language unification or major refactoring.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

Our evaluation verifies the correctness of distributed training, characterizes overhead and scalability to 256 ranks, and demonstrates two practical workflows: online training and online inference for lossy state compression. Future work will expand systematic comparisons across training schedules and comparable applications, explore bounded-memory streaming datasets, and integrate external data services to enable concurrent training across ensembles. More broadly, this orchestration provides a pragmatic bridge between parallel simulation and modern ML tooling, enabling adaptive, data-driven HPC applications and more responsive simulation environments.

Acknowledgments. This research used resources of the Argonne Leadership Computing Facility, a US DOE Office of Science user facility at Argonne National Laboratory, and is based on research supported by the US DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

1. Ahrens, J., Geveci, B., Law, C., Hansen, C., Johnson, C., et al.: Paraview: An end-user tool for large-data visualization. The Visualization Handbook **717**, 50038–1 (2005)
2. Ahrens, J., Jourdain, S., O'Leary, P., Patchett, J., Rogers, D.H., Petersen, M.: An image-based approach to extreme scale in situ visualization and analysis. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 424–434. IEEE (2014)
3. Atzori, M., Köpp, W., Chien, S.W., Massaro, D., Mallor, F., Peplinski, A., Rezaei, M., Jansson, N., Markidis, S., Vinuesa, R., et al.: In situ visualization of large-scale turbulence simulations in Nek5000 with Paraview Catalyst. The Journal of Supercomputing **78**(3), 3605–3620 (2022)
4. Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., Mauldin, J.: Paraview catalyst: Enabling in situ data analysis and visualization. In: Proceedings of the First Workshop on in situ infrastructures for enabling extreme-scale analysis and visualization. pp. 25–29 (2015)
5. Balin, R., Simini, F., Simpson, C., Shao, A., Rigazzi, A., Ellis, M., Becker, S., Doostan, A., Evans, J.A., Jansen, K.E.: In situ framework for coupling simulation and machine learning with application to CFD. arXiv preprint arXiv:2306.12900 (2023)
6. Bedrunka, M.C., Wilde, D., Kliemank, M., Reith, D., Foysi, H., Krämer, A.: Lettuce: Pytorch-based lattice Boltzmann framework. In: International Conference on High Performance Computing. pp. 40–55. Springer (2021)
7. Chen, X., Yang, G., Yao, Q., Nie, Z., Jiang, Z.: A compressed lattice Boltzmann method based on ConvLSTM and ResNet. Computers & Mathematics with Applications **97**, 162–174 (2021)
8. Gonzalez-Sieiro, J., Pardo, D., Nava, V., Calo, V.M., Towara, M.: Reducing spatial discretization error on coarse CFD simulations using an OpenFOAM-embedded deep learning framework. Engineering with Computers pp. 1–22 (2024)
9. Harrison, C., Larsen, M., Ryujin, B.S., Kunen, A., Capps, A., Privitera, J.: Conduit: a successful strategy for describing and sharing data in situ. In: 2022 IEEE/ACM International Workshop on in Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV). pp. 1–6. IEEE (2022)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9

10. Imambi, S., Prakash, K.B., Kanagachidambaresan, G.: Pytorch. In: Programming with TensorFlow: solution for edge computing applications, pp. 87–104. Springer (2021)
11. Jasak, H.: Openfoam: Open source CFD in research and industry. International Journal of Naval Architecture and Ocean Engineering **1**(2), 89–94 (2009)
12. Kurz, M., Offenhäuser, P., Viola, D., Resch, M., Beck, A.: Relexi—a scalable open source reinforcement learning framework for high-performance computing. Software Impacts **14**, 100422 (2022)
13. Larsen, M., Brugger, E., Childs, H., Harrison, C.: Ascent: A flyweight in situ library for exascale simulations. In: In Situ Visualization for Computational Science, pp. 255–279. Springer (2022)
14. Martin, A., Yousef, A., Liu, G., Ladd, W., Georgiadou, A., Stoop, J., Randles, A.: Performance portability evaluation of fluid-structure interaction simulations on heterogeneous platforms. In: ISC High Performance 2025 Research Paper Proceedings (40th International Conference). pp. 1–11. Prometeus GmbH (2025)
15. Maulik, R., Fytanidis, D.K., Lusch, B., Vishwanath, V., Patel, S.: PythonFOAM: In-situ data analyses with OpenFOAM and Python. Journal of Computational Science **62**, 101750 (2022)
16. Maulik, R., Sharma, H., Patel, S., Lusch, B., Jennings, E.: Deploying deep learning in OpenFOAM with tensorflow. In: AIAA Scitech 2021 Forum. p. 1485 (2021)
17. Pegolotti, L., Pfaller, M.R., Rubio, N.L., Ding, K., Brufau, R.B., Darve, E., Marsden, A.L.: Learning reduced-order models for cardiovascular simulations with graph neural networks. Computers in Biology and Medicine **168**, 107676 (2024)
18. Randles, A.P., Kale, V., Hammond, J., Gropp, W., Kaxiras, E.: Performance analysis of the lattice Boltzmann model beyond Navier-Stokes. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 1063–1074. IEEE (2013)
19. Sun, X., Cao, W., Shan, X., Liu, Y., Zhang, W.: A generalized framework for integrating machine learning into computational fluid dynamics. Journal of Computational Science **82**, 102404 (2024)
20. Tiwari, S.: The rise of intelligent machines: An introduction to artificial intelligence. Artificial Intelligence and Machine Learning in Drug Design and Development pp. 1–22 (2024)
21. Vardhan, M., Tanade, C., Chen, S.J., Mahmood, O., Chakravartti, J., Jones, W.S., Kahn, A.M., Vemulapalli, S., Patel, M., Leopold, J.A., et al.: Diagnostic performance of coronary angiography derived computational fractional flow reserve. Journal of the American Heart Association **13**(13), e029941 (2024)
22. Yokelson, D., Lappi, O., Ramesh, S., Väisälä, M.S., Huck, K., Puro, T., Norris, B., Korpi-Lagg, M., Heljanko, K., Malony, A.D.: Soma: Observability, monitoring, and in situ analytics for exascale applications. Concurrency and Computation: Practice and Experience **36**(19), e8141 (2024)
23. Yousef, A., Wetterer-Nelson, C., Han, M., et al.: Catalystml: In situ machine learning framework for parallel solvers. https://github.com/aymanzyy/catalystml (2026), gitHub repository
24. Yousef, A.Z., Draeger, E.W., Randles, A.: Low-cost post hoc reconstruction of HPC simulations at full resolution. In: 2023 IEEE 13th Symposium on Large Data Analysis and Visualization (LDAV). pp. 17–21. IEEE (2023)
25. Öngün, C.: Point cloud autoencoder. Github Repository (2020), https://github.com/cihanongun/Point-Cloud-Autoencoder

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29924-6_9