DLHub: Simplifying Publication, Discovery, and Use of Machine Learning Models in Science

Zhuozhao Li[a,b], Ryan Chard[b], Logan Ward[b], Kyle Chard[a,b,c], Tyler J. Skluzacek[a], Yadu Babuji[b,c], Anna Woodard[a], Steven Tuecke[b,c], Ben Blaiszik[b,c], Michael J. Franklin[a], Ian Foster[a,b,c]

[a]*Department of Computer Science, University of Chicago, Chicago, IL, USA*
[b]*Data Science and Learning Division, Argonne National Laboratory, Argonne, IL, USA*
[c]*Globus, University of Chicago, Chicago, IL, USA*

Abstract

Machine Learning (ML) has become a critical tool enabling new methods of analysis and driving deeper understanding of phenomena across scientific disciplines. There is a growing need for "learning systems" to support various phases in the ML lifecycle. While others have focused on supporting model development, training, and inference, few have focused on the unique challenges inherent in science, such as the need to publish and share models and to serve them on a range of available computing resources. In this paper, we present the Data and Learning Hub for science (DLHub), a learning system designed to support these use cases. Specifically, DLHub enables publication of models, with descriptive metadata, persistent identifiers, and flexible access control. It packages arbitrary models into portable servable containers, and enables low-latency, distributed serving of these models on heterogeneous compute resources. We show that DLHub supports low-latency model inference comparable to other model serving systems including TensorFlow Serving, SageMaker, and Clipper, and improved performance, by up to 95%, with batching and memoization enabled. We also show that DLHub can scale to concurrently serve models on 500 containers. Finally, we describe five case studies that highlight the use of DLHub for scientific applications.

Keywords: Learning Systems, Model Serving, Machine Learning, DLHub

1. Introduction

Learning systems are an important new class of systems designed to support the many phases of the Machine Learning (ML) lifecycle (see Figure 1). Various learning systems have been developed to support model development <1; 2>; scalable training across thousands of cores and GPUs <3>; model publication and sharing <4>; and low-latency and high-throughput inference <5>.

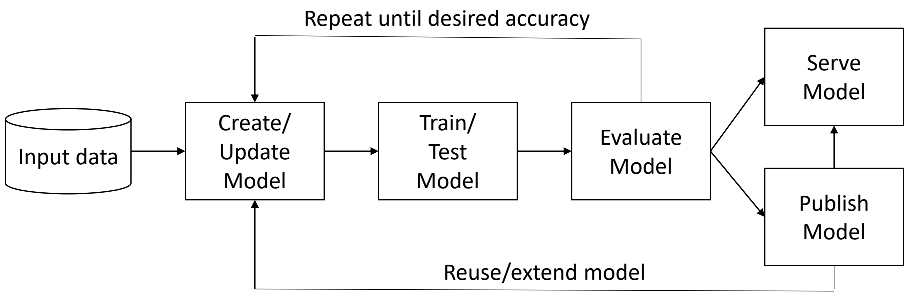


Figure 1: ML lifecycle, adapted from Miao et al. <6>

The rapid adoption of ML across science, for example to design and discover new materials and molecules <7; 8>; to detect and make cancer diagnoses earlier <9> and to enhance patient care <10>; acting as surrogates for more expensive simulations <11; 12>, guiding genome-editing capabilities <13>, brings with it unique and urgent challenges. For example, there is a need to discover, reuse, and reproduce models published in the literature to validate and extend cutting-edge results; publish models with descriptive metadata and persistent identifiers for discovery and unambiguous citation; and to scalably and reliably execute models on the myriad resources available to researchers.

In this paper, we present the Data and Learning Hub for science (DLHub) and outline initial experiences applying this learning system to science. While many learning systems focus on building and training ML models <14; 15; 3>, DLHub is a unique learning system that is designed to support the publication and serving of ML models in science. DLHub is implemented as a cloud-hosted service that allows researchers to deposit and share models of various types, including TensorFlow <15>, Keras <16>, PyTorch <17>, and Scikit-learn <18>. It defines common metadata schemas for describing these models and the parameters and other inputs used to invoke them. It also implements a rich access control model that allows users to publish their models privately, publicly, or with a select group of other users.

DLHub offers a unique model serving infrastructure that is capable of serving many different types of models on a range of distributed computing resources including clouds, clusters, and supercomputers. The serving infrastructure builds upon funcX <19; 20>—a distributed Function-as-a-Service platform developed specifically to support remote and distributed execution of functions. DLHub implements a flexible pipeline that converts deposited models into servables—executable containers that implement a standard DLHub execution interface, irrespective of the model type, and includes the trained model, model components (e.g., training weights, hyperparameters), and dependencies (e.g., system or Python packages). DLHub registers these published functions with funcX which then allows the servables to be transferred and deployed to remote computing resources and invoked one or more times on different input arguments. funcX elastically provisions compute nodes (e.g., via cloud API or batch scheduler) in response to workload requirements, deploys special funcX worker agents in servable containers for fine-grain execution, and then manages the secure and reliable execution of inference tasks.

In this paper, we extend our previous work <21; 22> by outlining the new DLHub architecture that is able to serve models on arbitrary distributed resources using funcX. This architecture also allows researchers to use their own resources when invoking models published in DLHub.

We evaluate the performance of DLHub by showing that it can scale to hundreds of concurrent containers when deployed on different resources including a supercomputer, cluster, and Kubternetes cluster. and compare it against alternative learning systems. We show that DLHub performs comparably with other systems, such as TensorFlow Serving <23> and SageMaker <3>, when using a Kubernetes cluster. Finally, we show that memoization and batching can significantly improve performance and that DLHub can serve models on remote computing resources in less than 75ms.

The remainder of the paper is structured as follows. In §2, we outline the need for, and unique requirements of, learning systems in science. In §3, we survey a range of model repository and serving systems. In §4, we present the DLHub architecture and describe how it supports publication and serving. In §5, we evaluate DLHub by exploring the serving latency and scalability, performance optimizations, and comparing it against three related systems. Finally, in §6, we present case studies that highlight the benefits of DLHub in science, and we summarize our contributions in §7.

2. Specialized Requirements of Science

Increasingly sophisticated learning systems are being developed, in particular by cloud providers, to support commercial ML use cases. However, scientific use of ML has specialized requirements, including the following.

Publication, citation, and reuse: The scholarly process is built upon a common workflow of publication, peer review, and citation. Progress is dependent on being able to locate, verify, and extend prior research, and careers are built upon publications and citation. As scholarly objects, ML models should be subject to similar publication, review, and citation models. Lacking standard methods for doing so, (a) many models associated with published literature are not available <24>; and (b) researchers adopt a range of ad hoc methods (from customized websites to GitHub) for sharing ML models <25; 26; 27>.

Reproducibility: Concerns about reproducibility are having a profound effect on research <28>. While reproducibility initiatives have primarily focused on making data and experimental processes available to reproduce findings, there is a growing interest in making computational methods available as well <29; 30; 31>.

Unlike sharing software products, there is little guidance for sharing ML models and their artifacts (e.g., weights, hyper-parameters, and training/test sets). Without publishing these artifacts, it is almost impossible to verify or build upon published results. Thus, there is a growing need to develop standard ML model packages and metadata schema, and to provide rich model repositories and serving platforms that can be used to reproduce published results.

Research infrastructure: While industry and research share common requirements for scaling inference, the execution landscape differs. Researchers often want to use multiple (often heterogeneous) parallel and distributed computing resources to develop, optimize, train, and execute models. Examples include: laboratory computers, campus clusters, national cyberinfrastructure (e.g., XSEDE <32>, Open Science Grid <33>), supercomputers, and clouds. They often have their own resources that they would like to use for inference. Thus, learning systems need to support execution on different resources and enable migration between resources.

Scalability: Large-scale parallel and distributed computing environments enable ML models to be executed at unprecedented scale. Researchers require learning systems that simplify training and inference on enormous scientific datasets and that can be parallelized to exploit large computing resources.

Low latency: ML is increasingly being used in real-time scientific pipelines, for example to process and respond to events generated from sensor networks; classify and prioritize transient events from digital sky surveys for exploration; and to perform error detection on images obtained from X-ray light sources. There is a need in each case for low latency, near real-time ML inference for anomaly/error detection and for experiment steering purposes. As both the number of devices and data generation rates continue to grow, there is also a need to be able to execute many inference tasks in parallel, whether on centralized or "edge" computers.

Research ecosystem: Researchers rely upon a large and growing ecosystem of research-specific software and services: for example, Globus <34> to access and manage their data; community and institution-specific data sources (e.g., the Materials Data Facility <35> and Materials Project <36>) as input to their ML models; and research authentication and authorization models (e.g., using campus or ORCID identities).

Model in the loop: Scientific analyses often involve multiple steps, such as the staging of input data for pre-processing and normalization, extraction of pertinent features, execution of one or more ML models, application of uncertainty quantification methods, post-processing of outputs, and recording of provenance. There is a growing need to expose models with simple and secure Web interfaces for on-demand consumption such that models can be used in cohesive, shareable, and reusable scientific workflows.

3. Learning Systems: A Brief Survey

We define a learning system as "a system that supports any phase of the ML model lifecycle including the development, training, inference, sharing, publication, verification, and reuse of a ML model." To elucidate the current landscape, we survey a range of existing systems, focusing on those that provide model repository and serving capabilities. Model repositories catalog collections of models, maintaining metadata for the purpose of discovery, comparison, and use. Model serving platforms facilitate online model execution.

A repository or serving platform may be provided as a *hosted service*, in which case models are deployed and made available to users via the Internet, or *self-service*, requiring users to operate the system locally and manage the deployment of models across their own infrastructure.

3.1. Model Repositories

Model repositories catalog and aggregate models, often by domain, storing trained and untrained models with associated metadata to enable discovery and citation. Metadata may be user-defined and/or standardized by using common publication schemas (e.g., author, creation date, description, etc.) and ML-specific schemas. ML-specific metadata include model-specific metadata (e.g., algorithm, software version, network architecture), development provenance (e.g., versions, contributors), training metadata (e.g., datasets and parametrization used for training), and performance metadata (e.g., accuracy when applied to benchmark datasets). Model repositories may provide the ability to associate a persistent identifier (e.g., DOI) and citation information such that creators may receive credit for their efforts.

Table 1 summarizes four representative model repositories plus DLHub along the following dimensions; we describe each repository in more detail below.

- **Publication and curation:** Whether models can be contributed by users and if any curation process is applied.
- **Domain:** Whether the repository is designed for a single domain (e.g., bioinformatics) or for many domains.
- **Model types:** What types of ML models can be registered in the repository (e.g., any model type, TensorFlow).
- **Data integration:** Whether data (e.g., training/test datasets) and configuration (e.g., hyperparameters) can be included with the published model.
- **Model metadata:** Whether the repository supports publication of model-specific metadata, model building requirements, and/or invocation metadata.
- **Search capabilities:** What search mechanisms are provided to allow users to find and compare models.
- **Model versioning:** Whether the repository facilitates versioning and updates to published models.
- **Export**: Whether the repository allows models to be exported, and if so, in what format.

ModelHub <6> is a deep learning model lifecycle management system focused on managing the data artifacts generated during the deep learning life-

Table 1: Model repositories compared and contrasted. BYO = bring your own.

	ModelHub	**Caffe Zoo**	**ModelHub.ai**	**Kipoi**	**DLHub**
Publication method	BYO	BYO	Curated	Curated	BYO
Domain(s) supported	General	General	Medical	Genomics	General
Datasets included	Yes	Yes	No	No	Yes
Metadata type	Ad hoc	Ad hoc	Ad hoc	Structured	Structured
Search capabilities	SQL	None	Web GUI	Web GUI	Elasticsearch
Identifiers supported	No	BYO	No	BYO	BYO
Versioning supported	Yes	No	No	Yes	Yes
Export method	Git	Git	Git/Docker	Git/Docker	Docker

cycle, such as parameters and logs, and understanding the behavior of the generated models. Using a Git-like command line interface, users initialize repositories to capture model information and record the files created during the creation process. Users then exchange a custom-built model versioning repository, called DLV, through the hosted service to enable publication and discovery. ModelHub is underpinned by Git, inheriting versioning capabilities, support for arbitrary datasets, scripts, features, and accommodates models regardless of domain. A custom SQL-like query language, called DQL, allows ModelHub users to search across repositories on characteristics such as authors, network architecture, and hyper-parameters.

Caffe Model Zoo <37> is a community-driven effort to publish and share Caffe <14> models. Users contribute models via Dropbox or GitHub Gists. The Model Zoo provides a standard format for packaging, describing, and sharing Caffe models. It also provides tools to enable users to upload models and download trained binaries. The Model Zoo operates a community-edited Wiki page to describe each of the published models, aggregating information regarding manuscripts, citation, and usage documentation in an unstructured format. The project encourages open sharing of models, trained weights, datasets, and code through GitHub. The Model Zoo provides guidelines on how to contribute models and what metadata should be included in the accompanying *readme.md* file without enforcing a specific schema. Users typically include citation information, links to the project page, a GitHub address for the model's code, and in some cases, a link to `haystack.ai` where the model can be tested.

ModelHub.ai <38> is a service to crowdsource and aggregate deep learning models related to medical applications. ModelHub.ai has a Web interface that lets users review published models, experiment with example inputs, and even test them online using custom inputs. The service provides detailed documentation and libraries to package models into a supported Docker format. Once packaged, users can add the model and any associated metadata to the ModelHub GitHub repository and submit a pull-request. The contributed model is curated and added to the catalog. The ModelHub.ai project provides both a Flask and Python API to interact with Dockerized models, which can be retrieved by either downloading the Docker image or cloning the GitHub repository.

Kipoi <4> is a repository of trained models for genomics that includes more than 2000 models of 21 different types. It provides a command line interface (CLI) for publishing and accessing models. On publication, the CLI prompts the user for descriptive metadata and generates a configuration file containing the metadata needed to discover and run the model. Users can then publish their models by submitting a pull-request to the Kipoi GitHub repository. Models can be listed and retrieved through the API and then invoked locally.

3.2. Model Serving

ML model serving platforms provide on-demand model inference. Existing model serving platforms vary in both their goals and capabilities: for example, some focus on serving a specific type of model with extremely low latency, while others prioritize ease of use and simple inference interfaces. We have identified the following important dimensions to capture the differences between model serving platforms. Table 2 summarizes popular model serving platforms plus DLHub along these dimensions.

- **Service model:** Whether the platform is offered as a hosted service or requires self-service deployment.
- **Model types:** What languages and types are supported (e.g., TensorFlow, Scikit-learn, R, Python, etc.).
- **Input types:** The range of input types supported by the system (e.g., structured, files, or primitive types).
- **Training capabilities:** Whether the system supports training.
- **Transformations:** Whether pre-/post-processing steps can be deployed.
- **Invocation interface:** What methods of interaction with the models are supported.
- **Execution environment:** Where models are deployed (e.g., cloud, Kubernetes, Docker).

PennAI <39> provides model serving capabilities for biomedical and health data. The platform allows users to apply six ML algorithms, including regressions, decision trees, SVMs, and random forests to their datasets, and to perform supervised classifications. The PennAI website provides a user-friendly interface for selecting, training, and applying algorithms to data. The platform also exposes a controller for job launching and result tracking, result visualization tools, and a graph database to store results. PennAI does not support user-provided models, but does provide an intuitive mechanism to train classification tools and simplify the integration of ML into scientific processes.

Table 2: Serving systems compared and contrasted. K8s = Kubernetes.

	PennAI	**TensorFlow Serving**	**Clipper**	**SageMaker**	**Algorithmia**	**DLHub**
Service model	Hosted	Self-service	Self-service	Hosted	Hosted	Hosted
Model types	Limited	TensorFlow Servables	General	General	General	General
Input types supported	Unknown	Primitives, Files	Primitives	Structured, Files	Unknown	Structured, Files
Training supported	Yes	No	No	Yes	No	No
Transformations	No	Yes	No	No	No	Yes
Invocation interface	Web GUI	gRPC, REST	gRPC, REST	gRPC, REST	API, REST	API, REST
Execution environment	Cloud	Docker, K8s, Cloud	Docker, K8s	Cloud, Docker	Docker, K8s, Cloud	Docker, K8s, Singularity, Cloud

TensorFlow Serving <23> is the most well-known model serving solution and is used extensively in production environments. TensorFlow Serving provides high performance serving via gRPC and REST APIs and is capable of simultaneously serving many models, with many versions, at scale. TensorFlow Serving provides the lowest latency serving of any of the surveyed platforms. It serves trained TensorFlow models using the standard `tensorflow_model_server`, which is built in C++. Although TensorFlow Serving does support a range of model types—those that can be exported into TensorFlow servables—it is limited in terms of its support for custom transformation codes and does not support the creation of pipelines between servables. TensorFlow Serving is also self-service, requiring users to deploy and operate TensorFlow Serving on local (or cloud) infrastructure in order to deposit models and perform inferences. TensorFlow also provides model repository capabilities through a library of reusable ML modules, called TensorFlow Hub.

Clipper <5> is a prediction serving system that focuses on low latency serving. It deploys models as Docker containers, which eases management complexity and allows each model to have its own dependencies wrapped in a self-contained environment. Clipper includes several optimizations to improve serving performance including data batching and memoization. Clipper also provides a model selection framework to improve prediction accuracy. However, because Clipper needs to Dockerize the models on the manager node, it requires privileged access, which is not available on all execution environments (e.g., high performance computing clusters).

SageMaker <3> is an ML serving platform provided by Amazon Web Services that supports both the training of models and the deployment of trained models as Docker containers for serving. It helps users to handle large data efficiently by providing ML algorithms that are optimized for distributed environments. SageMaker APIs allow users to deploy a variety of ML models and integrate their own algorithms. In addition, trained models can be exported as Docker containers for local deployment.

Algorithmia <40> is an industry platform for deploying and scaling ML models in the cloud for production use. Algorithmia creates a GitHub repository for each published model before containerizing it with Docker and making it available via the Internet. Model deployments are elastically scalable and can leverage accelerators, enabling users to tune performance. Algorithmia provides users with fine-grained access control for sharing and disseminating their contributions. While many of Algorithmia's capabilities overlap with DLHub, we provide additional capabilities to enrich scientific ML, such as making models citable, reusable, and deployable on existing research computing infrastructures.

Kubeflow <41> is a collection of open source ML services that can be deployed to provide a fully-functional ML environment on a Kubernetes cluster. The system simplifies the deployment of various ML tools and services, including those to support model training, hyper-parameter tuning, and model serving. Kubeflow also integrates Jupyter Notebooks to provide a user-friendly interface to many of these services. Model serving in Kubeflow uses TensorFlow Serving, so we have not included it in our summary table.

4. DLHub Architecture and Implementation

DLHub is a learning system that provides model publishing and serving capabilities for scientific ML. DLHub's model repository supports user-driven publication, citation, discovery, and reuse of ML models from a wide range of domains. It offers rich search capabilities to enable discovery of, and access to, published models. DLHub automatically converts each published model into a "servable"—an executable DLHub container that implements a standard execution interface and comprises a complete model package that includes the trained model, model components (e.g., training weights, hyperparameters), and any dependencies (e.g., system or Python packages). DLHub can then "serve" the model by deploying and invoking one or more instances of the servable on execution site(s). DLHub provides high throughput and low-latency model serving by dispatching tasks in parallel to the remote execution site(s). DLHub implements a flexible inference system, built upon the funcX distirbuted function as a service platform <19>, via which inference tasks can be executed on arbitrary computing resources. The DLHub architecture, shown in Figure 2, comprises two core components: the *Management Service and Catalog* and the distributed inference execution system.

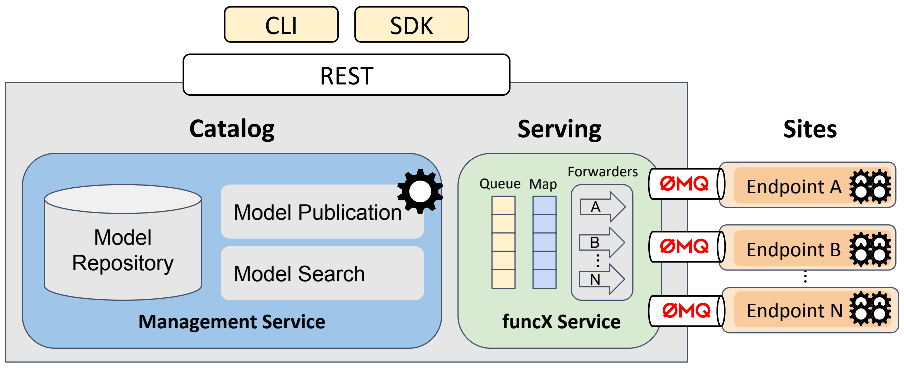


Figure 2: DLHub architecture. User requests, submitted via REST, SDK, or CLI (top), can result in model publication to a catalog or the dispatch of serving requests to servables deployed on any computing resources with a funcX endpoint (right).

4.1. Management Service and Catalog

DLHub's Management Service is the user-facing interface to DLHub. It enables users to publish models and query available models. The Management Service includes functionality to build models and optimize task performance.

Model repository: The primary function of the Management Service is to support the publication and discovery of models. DLHub defines a general model schema that is used to describe all published models. The schema includes standard publication metadata (e.g., creator, date, name, description) as well as ML-specific metadata such as model type (e.g., Keras, TensorFlow) and input and output data types. These metadata are registered with a search catalog to enable flexible discovery.

Model discovery: DLHub's discovery interface supports fine-grained, access-controlled search across registered model metadata. It provides a rich search model, in which model metadata can be queried using free text queries, partial matching, range queries, faceted search, and more through both the DLHub CLI and SDK.

Model publication: In order to provide a common model execution interface irrespective of model type, DLHub converts all published models into executable servables. DLHub provides two publication pipelines for users to publish models: either by directly uploading model components as well as descriptive metadata for building the servable to the DLHub service, or importing from a GitHub repository and uploading minimal DLHub-specific metadata (e.g., model type and entry point) defined in a DLHub-specific JSON file. In both cases the servable is uploaded to the model repository and any user-supplied metadata is registered alongside the servable location in the search index.

Implementation: DLHub supports several publication modes, for example, a user can choose to 1) upload model components directly to DLHub; 2) specify a remote location on an AWS S3 bucket or Globus endpoint; or 3) provide a GitHub location for DLHub to clone. Irrespective of the publication mode, the Management Service access, transfers, or downloads the published model components and builds a servable in a DLHub-compatible format. DLHub first uses repo2docker to build a base container environment, incorporating all user-defined dependencies. It then uses the base container to build a servable by adding DLHub-specific components and interfaces. DLHub uses Amazon Elastic Container Registry as the servable repository. The metadata for each published servable is indexed in a Globus Search catalog <42>. Globus Search is built upon Elasticsearch, providing the flexibility to accommodate diverse scientific metadata.

4.2. Model Descriptions

We have developed a metadata schema to describe published models and encode authorship, provenance, dependency, and interface metadata. Our schema achieves two key goals: first, it enables model developers to describe the information necessary for others to use their model, such as its inputs, training configuration, and outputs; and second, it encodes the "recipe" for DLHub to create, deploy, and serve the model.

The schema first encodes information necessary for humans to understand and use a model (e.g., inputs/outputs). It must include the type of model (e.g., a TensorFlow model) and any configuration information that is required (e.g., the model's weights file).

The model description also includes metadata needed for DLHub to construct the servable. Specifically, it defines the computational environment, software dependencies, and associated files necessary to create, deploy, and execute a servable. When defining the environment and dependencies, a model developer can list the necessary Python packages, configuration files (e.g., a model's weights file or license information), or repo2docker <43> configuration files.

Implementation: The DLHub model description schema is publicly accessible and builds upon prior work and standards. We use the DataCite <44> metadata schema to describe provenance, ownership, references to associated artifacts (e.g., code repositories, publications, and datasets), human-readable descriptions of the model, and persistent identifiers.

DLHub provides tools to simplify the description process by automatically extracting information from the environment (e.g., Python package versions) and from the model implementation. For example, the DLHub tools can identify the types and shapes of a Keras model's inputs/outputs that are stored within the HDF5 model file. The DLHub SDK reads this information from the HDF5 file, enabling the creation of a valid model definition in just three lines of code. The DLHub SDK provides similar tools to help users generate DataCite-compatible metadata, define the computational environment, and describe other types of models.

4.3. Inference Execution System

DLHub coordinates the execution of inference tasks on remote resources. This architecture focuses on high performance and low latency model inference as well as flexiblity in terms of where inference tasks are executed. Specifically, DLHub allows researchers to execute inference tasks onKubernetes clusters, HPC resources, or clouds using various container technogies (e.g., Docker, Singularity or Shifter), on edge devices, or even on their own execution resources using any of these containerization mechanisms.

DLHub supports both synchronous and asynchronous task execution. In asynchronous mode, the DLHub SDK returns a task UUID that can be used subsequently to monitor the status of the task and retrieve its result.

Implementation: DLHub's on-demand inference is built on the funcX platform <20>. funcX is a distributed Function-as-a-Service (FaaS) platform inspired by the original DLHub model. We briefly describe funcX and outline how it is used by DLHub.

funcX: funcX enables the managed execution of functions—snippets of Python code—on arbitrary remote resources. Users can register and discover functions through a cloud-hosted service and then execute those functions with arbitrary input parameters on arbitrary *endpoints*. Where an endpoint abstracts a specific compute resource, whether a single edge device or a supercomputer, in a manner defined by the funcX agent software.

The funcX service implements a secure, low-latency task execution model with hierarchical queues for reliability. Tasks are submitted to the funcX Web service where they are queued for execution. A Python *Forwarder* process is operated for each endpoint. The Forwarder retrieves tasks from the cloud-hosted queues and transmits them to the endpoint via a secure, low-latency, and reliable message communication channel. Once delivered to the endpoint, tasks are internally queued until they can be scheduled for execution on the resource. Results are returned via the same channel and deposited in a result queue until they can be retrieved by the user. funcX uses a Redis store to implement the cloud-based queues. Redis is an easy-to-scale, in-memory key-value store. Each function execution request is stored in a Redis hashmap and the task identifier is added to the endpoint's queue. funcX uses ZeroMQ to establish high performance communication channels between the forwarder and endpoint.

Endpoints: A funcX endpoint is a logical representation of a computational resource. The corresponding funcX agent deployed on that resource implements an API that allows the funcX service to dispatch functions for execution. The agent handles authentication and authorization, provisioning of nodes on the compute resource, transfer of function containers, staging of data, and monitoring and management. Administrators or users can deploy an endpoint agent and register an endpoint for themselves and/or others, providing descriptive (e.g., name, description) and execution (e.g., container technology, scheduler) metadata. Each endpoint is assigned a unique identifier for subsequent use. The funcX endpoint thus serves for computation a comparable role in the research CI ecosystem to that served by the Globus endpoint for data.

The funcX agent uses the Parsl <45; 46> execution model to dynamically acquire compute nodes, deploy servable containers, and execute functions on those deployed containers. Parsl implements a modular execution model that supports various common cluster and supercomputer schedulers as well as common cloud computing providers. In each case, it uses platform-specific mechanisms to requests nodes, deploy pilot job software, dispatch tasks to workers, monitor progress, and report on results. On a Kubernetes cluster, for example, the endpoint creates a Kubernetes Deployment consisting of n pods for each servable that is to be executed, a number configurable in the Management Service. The funcX agent then deploys worker engines in each servable container which connect back to retrieve execution requests. The funcX agent dispatches requests to the appropriate containers, load balancing them automatically across the available pods.

DLHub and funcX: When a user publishes a model to DLHub, we create and register a function with funcX and associate it with the DLHub servable container. This allows funcX to deploy the servable on-demand to perform DLHub invocations. When a user invokes a servable using DLHub the request is routed to a DLHub-operated funcX endpoint. The funcX agent will then deploy the servable and, once the servable is ready, deliver the request for execution. The funcX agent is responsible for deploying and managing servables, monitoring incoming requests from DLHub (via the funcX service), and then executing waiting tasks. The funcX agent can be deployed in Docker environments, Kubernetes clusters, HPC resources via Singularity or Shifter, or locally via any of these containerization mechanisms.

4.4. Security

DLHub implements a comprehensive security model to ensure that all operations are performed by authenticated and authorized users. DLHub's security model allows users to authenticate using one of hundreds of supported identity providers (e.g., campus, ORCID, +Google). When authenticating, the Management Service first validates the user's identity, and then retrieves short-term access tokens that allow it to obtain profile information about the user, to access/download data on their behalf, and to compute inference tasks on their behalf. These capabilities allow DLHub to precomplete publication metadata using profile information and also to transfer model components and inputs from arbitrary locations.

DLHub relies on container technology to provide secure execution sandboxes for inference isolation, ensuring inference tasks cannot interfere with other tasks and can only access data and computing resources within the specified context.

For model reliability, DLHub stores hashes of published models to ensure the integrity of models executed both within our serving infrastructures and when deployed locally. However, this approach does not protect users against downloading models published maliciously. In this case, users must determine the safety of the model by trusting the author or manually inspecting the servable code before execution. Importantly, DLHub exposes the publisher's authenticated identity (in most cases an institution identity) and thus makes it simple to verify the author. We are actively explore methods to further validate the function of servables. We intend to encourage users to follow the Findable, Accessible, Interoperable, and Reusable (FAIR) principles <47> when they publish their models to DLHub. We also aim to investigate developing a community-driven governance system in which users can review and recommend servables.

Implementation: DLHub uses Globus Auth <48> for authentication and authorization. Globus Auth is a flexible identity and access management service that is designed to broker authentication and authorization decisions between users, identity providers, resource serves, and clients. The DLHub Management Service is registered as a Globus Auth resource server with an associated scope for programmatic invocation. funcX, and funcX endpoints, are also registered as independent Globus Auth resource servers, enabling secure routing of inference tasks to compute endpoints.

4.5. DLHub Interfaces

DLHub offers a REST API, a Python Software Development Kit (SDK), and a Command Line Interface (CLI) for publishing, managing, and invoking models.

The DLHub Python **SDK** supports programmatic construction of JSON documents that specify publication and model-specific metadata that complies with DLHub-required schemas. The SDK can then be used to publish the model by uploading the JSON metadata documents. The SDK also supports programmatic interactions with DLHub to discover, update, and invoke published models.

The DLHub **CLI** provides an intuitive Git-like user interface to interact with DLHub. It provides commands for initializing a DLHub servable in a local directory, publishing the servable to DLHub, creating metadata using the SDK, and invoking the published servable with input data.

5. Evaluation

To evaluate DLHub we conducted experiments to explore its serving performance, the impacts of memoization and data batching, and scalability on different compute resources. We also compared its serving performance against TensorFlow Serving, Clipper, and SageMaker.

5.1. Experimental Setup

Platforms. We deployed funcX endpoints on three different clusters: Theta <49>, Cooley <50>, and PetrelKube <51> at Argonne National Laboratory. Theta is a supercomputer with 4392 nodes, each containing a 64-core Intel Xeon Phi "Knights Landing" (KNL) processor, 16 GB MCDRAM, 192 GB of DDR4 RAM, and interconnected with high speed InfiBand. Cooley is a cluster designed for data analysis. It has 126 computing nodes, each equipped with 12 CPU cores, one NVIDIA Tesla K80 dual-GPU, 384 GB RAM per node and Infiniband interconnect. PetrelKube is a 14-node Kubernetes cluster, each containing two E5-2670 CPUs, 128GB RAM, and 40GbE network interconnect.

Servables. We use six servables in our evaluation.

The first is a baseline "**no-op**" task that returns "hello world" when invoked.

The second is Google's 22-layer Inception-v3 <52> model ("**Inception**"). Inception is trained on a large academic dataset for image recognition and classifies images into 1000 categories. Inception takes an image as input and outputs the five most likely categories.

The third is a multi-layer convolutional neural network trained on CIFAR-10 <53> ("**CIFAR-10**"). This common benchmark problem for image recognition takes a 32×32 pixel RGB image as input and classifies it in 10 categories.

The final three servables are part of a workflow used to predict the stability of a material given its elemental composition (e.g., NaCl). The model is split into three servables: parsing a string with pymatgen <54> to extract the elemental composition ("**matminer_util**"), computing features from the element fractions by using Matminer <55> ("**matminer_featurize**"), and executing a scikit-learn random forest model to predict stability ("**matminer_model**"). The model was trained with the features of Ward et al. <7> and data from the Open Quantum Materials Database <56>.

We create both Singularity and Docker containers for all these six servables to be served on Theta (Singularity), Cooley (Singularity), and Kubernetes (Docker).

5.2. Experiments

To remove bias we disable DLHub memoization mechanisms and restrict data batch size to one in the following experiments, except where otherwise noted.

5.2.1. Latency

We study the latency imposed by each component in DLHub. To do so, we deployed the "no-op" servable to a warmed container on the Cooley cluster, and instrumented each system component to record start and finish times. Figure 3 shows the breakdown of total execution time at the funcX web service, forwarder, endpoint, and worker, in addition to network transport times. We see that of the total 72ms round-trip time, the "no-op" function spends a combined 54ms across all system components, and 18ms traversing the network. We observe that the overhead for executing the function on the worker is just 2ms. A majority of the system latency is due to queuing and authorization calls at the web service (37ms total), and when dispatching the servable request from the endpoint to the worker (15ms total). This implies that the DLHub inference execution system incurs minimal latencies.

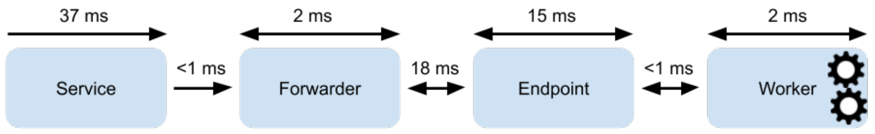
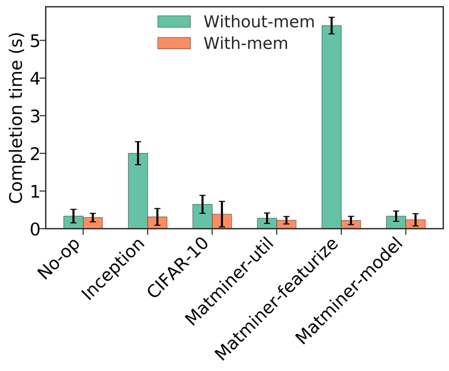


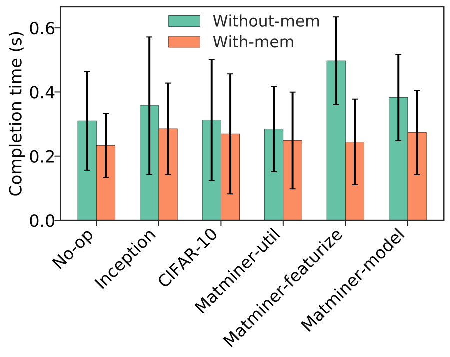
Figure 3: Latency breakdown for no-op DLHub servable on Cooley cluster.

5.2.2. Memoization

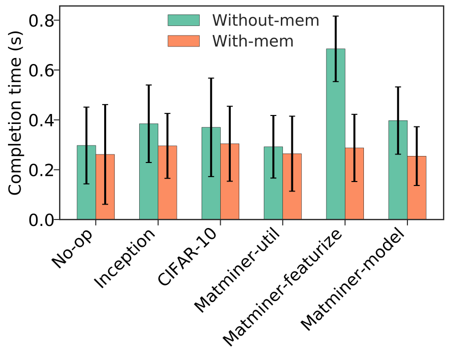
DLHub implements memoization <57>, at the funcX endpoint, to cache the inputs and outputs for each request and return the cached output for a new request if its inputs are in the cache. While memoization is not necessarily applicable to all ML models, we have found it useful in cases with repeated execution of deterministic models with the same arguments (e.g., content recommendation). To investigate the effect of memoization on serving performance we submitted requests with the same inputs, with memoization enabled and disabled. Figure 4 shows the total completion time of requests. We observe that memoization reduces total completion time by 10–95%, as it removes the time to compute the inference. The total completion time with memoization enabled for different servables are relatively consistent (around 0.2–0.4s, primarily due to network latencies and system overheads). The performance improvements differ between models due to the variance of computation time and input size.



(a) Caching on Theta.



(b) Caching on Cooley.



(c) Caching on PetrelKube.

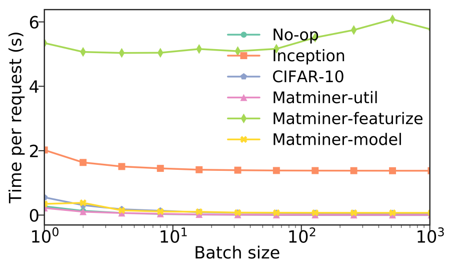
Figure 4: Performance impact of memoization on Theta, Cooley, and PetrelKube. Bars and error bars show mean and standard deviation.

5.2.3. Batching

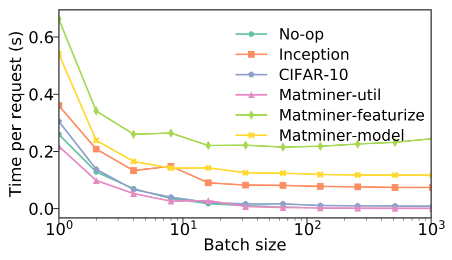
DLHub supports batching of requests to improve overall throughput by amortizing system overheads and network latencies over many requests. To study how performance varies with batch size we deployed funcX endpoints on Theta, Cooley and PetrelKube and limited the number of deployed containers of each servable to one. We then sent a batch of invocations to each servable and increased the batch size (i.e., the number of invocations in a single query) from 1 to 1024. Figure 5 shows the time per request versus the batch size. The time per request is computed as the total completion time of a batch divided by the batch size. We observe that all servables follow a similar trend: as the batch size increases, the time per request decreases. This is because batching amortizes system overheads and network latencies over many requests. The benefit of batching starts to diminish as the batch size becomes large. This is because the computation on each servable begins to dominate the total completion time. Similarly, the computation using Theta KNL nodes is relatively slow, causing the computation time to account for a greater percentage of the total completion time. The result is that batching is less beneficial on Theta than on other platforms. In future work, we intend to develop servable profiles and to explore adaptive batching algorithms that can intelligently distribute serving requests to reduce latency.

5.2.4. Scalability

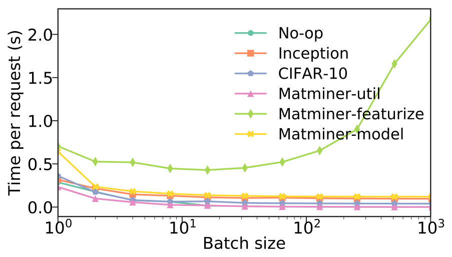
We evaluate the scalability of DLHub by sending inference requests to connected funcX endpoints. We report only the scalability of the endpoints as the other components are hosted on AWS services, which are known to be highly scalable. We deployed funcX endpoints on Argonne's Theta, Cooley, and PetrelKube computers with varying numbers of containers for each of the six servables. We set 64 and 12 containers per node on Theta and Cooley, respectively. On PetrelKube, Kubernetes will automatically manage the container deployment to nodes. We performed 1000 requests of each servable to the endpoints directly and measured the completion time of all requests. Figure 6 shows the



(a) Batching on Theta.



(b) Batching on Cooley.



(c) Batching on PetrelKube.

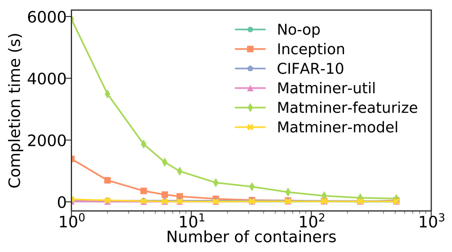
Figure 5: Time per request versus batch size on Theta, Cooley, and PetrelKube.

results on different platforms.

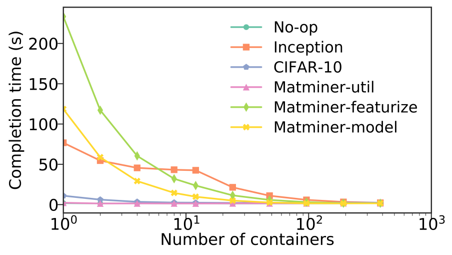
Our results show that the endpoint can easily deploy hundreds of containers for each servable. However, we also see that scalability is dependent on the servable itself. For example, when serving Matminer-featurize requests, throughput increases rapidly up to ∼32 containers, after which more containers have diminishing benefits and throughput is saturated, because task dispatch latencies dominate execution time. As expected, servables that execute for shorter periods of time (e.g., Matminer-model) show less benefit as additional containers are used, and vice versa.

5.2.5. Serving Comparison

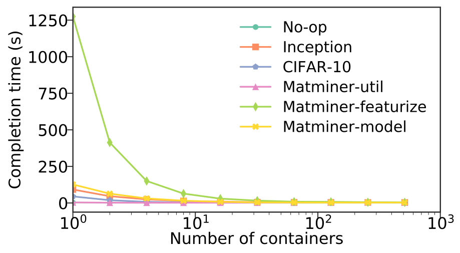
We used CIFAR-10 and Inception to compare the serving performance of TensorFlow Serving, SageMaker, Clipper, and DLHub when hosted on the PetrelKube Kubernetes cluster For TensorFlow Serving, we export the trained models and use the standard `tensorflow_model_server`. For SageMaker, we



(a) Scaling on Theta.



(b) Scaling on Cooley.



(c) Scaling on PetrelKube.

Figure 6: Scalability performance.

use the SageMaker service to create the models before exporting the model as a Docker container and deploy it as a Pod on PetrelKube. For Clipper, we use its Kubernetes container manager to deploy it on PetrelKube and register the CIFAR-10 and Inception models. For DLHub we use a funcX endpoint deployed on PetrelKube. To standardize our measurements we remove network overheads by submitting tasks directly to each platform and report the average time from 100 requests for each model and platform.

TensorFlow Serving provides two model serving APIs: REST and gRPC. SageMaker also supports serving TensorFlow models through TensorFlow Serving or its native Flask framework. In our experiments, we explore all possible APIs and frameworks, i.e., TFServing-REST, TFServing-gRPC, SageMaker-TFServing-REST, SageMaker-TFServing-gRPC, SageMaker-Flask. In addition, as Clipper supports caching, we evaluate it with and without caching.

Figure 7 shows the invocation times of CIFAR-10 and Inception using each serving system. We see that the servables invoked through the TensorFlow

Serving framework (i.e., TFServing-gRPC, TFServing-REST and SageMaker-TFServing) outperform those using other serving systems (SageMaker-Flask and DLHub). This is because the core tensorflow_model_server, implemented in C++, outperforms Python-based systems. gRPC leads to slightly better performance than REST due to the overhead of the HTTP protocol. Clipper performs similarly to other systems when caching is enabled. The clipper caching model maintains a cache at the query frontend (on a PetrelKube pod) and therefore performance is not significantly better than other systems as it requires that the request be transmitted to the query frontend. DLHub's performance is comparable to the other Python-based serving infrastructures.

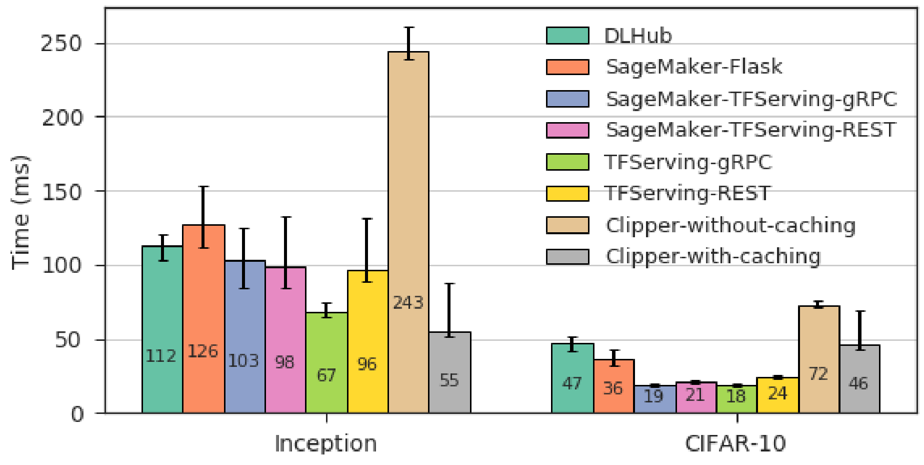


Figure 7: Performance of different serving systems on the Inception and CIFAR-10 problems.

6. Case Studies

To illustrate the value of DLHub we briefly outline five case studies that exemplify early adoption of the system.

6.1. Publication of Cancer Research Models

The Cancer Distributed Learning Environment (CANDLE) project <58> leverages leadership scale computing resources to address problems relevant to cancer research at different biological scales, specifically problems at the molecular, cellular, and population scales. CANDLE uses DLHub to securely share and serve a set of deep learning models and benchmarks using cellular level data to predict drug response based on molecular features of tumor cells and drug descriptors. As the models are still in development, they require substantial testing and verification by a subset of selected users prior to their general release. DLHub supports this use case by providing model sharing and discovery with fine grain access control. Thus, only permitted users can discover and invoke the models through the platform. Once models are determined suitable for general release, the access control on the model can be updated within DLHub to make them publicly available.

6.2. Enriching Materials Datasets

The Materials Data Facility <35; 59> (MDF) is a set of data services developed to enable data publication and data discovery in the materials science community. MDF allows researchers to distribute their data, which may be large or heterogeneous, and rapidly find, retrieve, and combine the contents of datasets indexed from across the community. MDF leverages several models published in DLHub to add value to datasets as they are ingested. When a new dataset is registered with MDF, automated workflows <42> are applied to trigger the invocation of relevant models to analyze the dataset and generate additional metadata. MDF extracts and associates fine-grained type information with each dataset which are closely aligned with the applicable input types described for each DLHub model.

6.3. Quality Control for SEM Images

DLHub is used by researchers studying the neuroanatomical structure of brains with a scanning electron microscope (SEM). Scientists produce films of thin brain slices and use the SEM to obtain images of the slices. A key limitation is the lack of built-in quality control mechanisms, and thus it is difficult to determine if the collected image is of suitable quality to be used in subsequent stitching and segmentation processes. These scientists use a set of models published in DLHub to automatically identify low quality images. Specifically they use five different focus detection models, and apply an ensemble methodology to determine if the image is of suitable quality.

6.4. X-Ray Tomography with Generative Adversarial Networks

X-ray computed tomography is a common imaging modality used at synchrotrons to understand the structure of materials. To avoid damaging samples, low-dose imaging with short exposure times are often used. Unfortunately, low-dose imaging can result in noisy measurements and low quality images. TomoGAN applies a Generative Adversarial Network (GAN) approach to improve the quality of 3D tomography images by reducing noise and eliminating artifacts. TomoGAN, a TensorFlow model, was easily published in DLHub by ingesting the *SavedModel* directory and generating a DLHub model description. The published TomoGAN model can be used by tomography researchers to process images uploaded via HTTPS. The resulting, denoised images are returned in a matter of minutes. The elastic scalability of DLHub allows researchers to trivially parallelize evaluation of each tomography frame. We are working with researchers to make the published model part of their research workflow.

6.5. Predicting Formation Enthalpy

DLHub makes it easy to link models plus pre- and post-processing transformations in pipelines to simplify the user experience. For example, a pipeline for predicting formation enthalpy from a material composition (e.g., SiO_2) can be organized into three steps: 1) conversion of material composition text into a Python object; 2) creation of a set of features, via matminer <55>, using the Python object as input; and 3) prediction of formation enthalpy using the matminer features as input. Once the pipeline is defined, the end user sees a simplified interface that allows them to input a material composition and receive a formation enthalpy. This and other more complex pipelines are defined as a series of modularized DLHub servables. Defining these steps as a pipeline allows data to be automatically passed between each servable, enabling execution to be performed server-side, drastically lowering both the latency and user burden to analyze inputs.

6.6. Evaluation

We have evaluated the invocation time and scalability of DLHub for each use case on PetrelKube. Figure 8 shows the distribution of invocation time for each use case, where x axis shows the invocation time and y axis shows the distribution of invocation time (computed by the number of occurrences divided by the total number of invocations). We see that each servable can be rapidly used with short execution times and that the distributions of the invocation times for all use cases except SEM are within a three-second range and relatively stable. The SEM use case exhibits a wider distribution than the other use cases due to variable network performance when downloading the input image to the container.

Figure 9 shows the scalability of each use case. We measured the completion times for 1000 concurrent invocations versus varying number of workers, for each of the five use cases with DLHub. We see that DLHub can scale each servable to more than 200 containers and that throughput is significantly increased in each case. We note that the benefit of scaling Formation and CANDLE use cases begins to diminish when using more than 200 containers. This is primarily because the constant system overheads (e.g., network and task dispatching latencies) start to dominate the completion times of Formation and CANDLE, as DLHub continues to scale. MDF presents a flatter trend than the other use cases due to the input data being downloaded from a single source and our experiments saturating the network connection.

6.7. Discussion

We briefly discuss the lessons learned by using DLHub in these five use cases. Prior to using DLHub, these use cases required a substantial amount of human effort to manually manage model versions, publish and share models

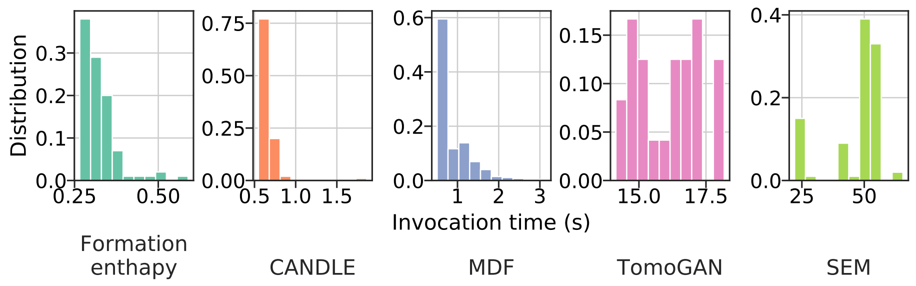


Figure 8: Distribution (computed by the number of occurrences divided by the total number of invocations) of invocation times for 100 invocations, for each of the five use cases with DLHub.

with others, deploy complex software environments on distributed computing resources, and reliably deploy models for real-time inferences at scale.

DLHub provides several benefits: First, DLHub manages different versions of the same model, removing challenges associated with tracking model versions and using incorrect versions. A key side effect of this is that researchers are able to deploy new versions of their models and compare the performance to any of the previously published versions. Second, DLHub's on-demand inference system abstracts the complexity of deploying models at different computing resources and enables researchers to easily deploy their models at scale, without requiring expert knowledge of batch submission interfaces and computing architectures. Finally, the containerization of models allows researchers to securely share models with others, removing the burden of porting models and environments to other locations.

7. Conclusion

The broad adoption of ML in science has necessitates the development of new learning systems to meet the unique requirements of science use cases. We have described one such learning system, DLHub, that is designed to address inefficiencies in two important phases of the ML lifecycle, namely the publication and serving of ML models plus associated data. DLHub's flexible publication

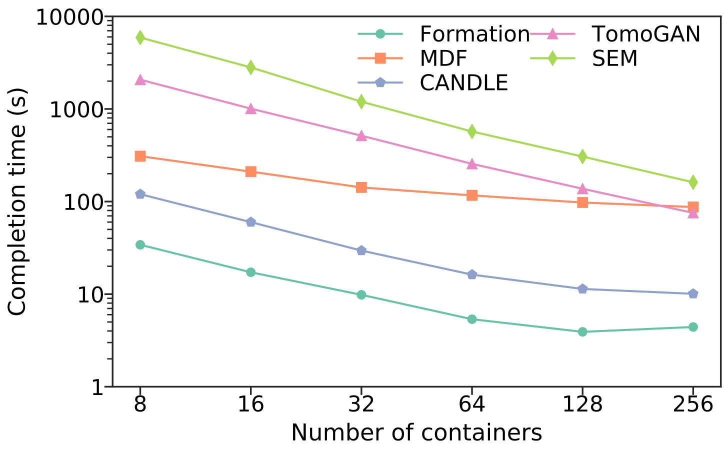


Figure 9: Completion times for 1000 concurrent invocations versus varying number of workers, for each of the five use cases with DLHub.

and sharing capabilities enable researchers to deposit and describe models using a common metadata schema to enable discovery. It also enables association of a persistent identifier for citation of published models. DLHub's scalable and low-latency model serving infrastructure enables the remote execution of containerized models on arbitrary computing resources. We showed that DLHub performs comparably with other serving platforms on Kubernetes clusters, while also enabling execution on clouds and clusters. We also showed that its memoization and batching optimizations can significantly improve serving performance.

Acknowledgments

This work was supported in part by Laboratory Directed Research and Development (LDRD) funding from Argonne National Laboratory and the RAMSES project, both from the U.S. Department of Energy under Contract DE-AC02-06CH11357, the Defense Advanced Research Projects Agency under grant number HR00111820006, and NSF under grant number 1550588. We thank Amazon Web Services for research credits and Argonne's Leadership Computing Facility and Joint Laboratory for System Evaluation for computing resources.

References

[1] P. Balaprakash, A. Tiwari, S. M. Wild, L. Carrington, P. D. Hovland, AutoMOMML: Automatic multi-objective modeling with machine learning, in: International Conference on High Performance Computing, Springer, 2016, pp. 219–239.

[2] Google Cloud AutoML, `https://cloud.google.com/automl/`. Accessed November 8, 2019.

[3] Amazon SageMaker, `https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html`. Accessed November 8, 2019.

[4] Z. Avsec, R. Kreuzhuber, J. Israeli, N. Xu, J. Cheng, A. Shrikumar, A. Banerjee, D. S. Kim, L. Urban, A. Kundaje, O. Stegle, J. Gagneur, Kipoi: Accelerating the community exchange and reuse of predictive models for genomics, bioRxiv 10.1101/375345 (2018).

[5] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, I. Stoica, Clipper: A low-latency online prediction serving system, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017, pp. 613–627.

[6] H. Miao, A. Li, L. S. Davis, A. Deshpande, Towards unified data and lifecycle management for deep learning, in: 33rd Intl Conf. on Data Engineering, IEEE, 2017, pp. 571–582.

[7] L. Ward, A. Agrawal, A. Choudhary, C. Wolverton, A general-purpose machine learning framework for predicting properties of inorganic materials, npj Computational Materials 2 (2016) 16028. `arXiv:1606.09551`, `doi:10.1038/npjcompumats.2016.28`.

[8] L. Ward, B. Blaiszik, I. Foster, R. S. Assary, B. Narayanan, L. Curtiss, Machine learning prediction of accurate atomization energies of organic molecules from low-fidelity quantum chemical calculations, arXiv preprint arXiv:1906.03233 (2019).

[9] K. Kourou, T. P. Exarchos, K. P. Exarchos, M. V. Karamouzis, D. I. Fotiadis, Machine learning applications in cancer prognosis and prediction, Computational and Structural Biotechnology Journal 13 (2015) 8 – 17. doi:https://doi.org/10.1016/j.csbj.2014.11.005.

[10] G. Simon, C. D. DiNardo, K. Takahashi, T. Cascone, C. Powers, R. Stevens, J. Allen, M. B. Antonoff, D. Gomez, P. Keane, et al., Applying artificial intelligence to address the knowledge gaps in cancer care, The oncologist 24 (6) (2019) 772–782.

[11] A. Z. Guo, E. Sevgen, H. Sidky, J. K. Whitmer, J. A. Hubbell, J. J. de Pablo, Adaptive enhanced sampling by force-biasing using neural networks, The Journal of chemical physics 148 (13) (2018) 134108.

[12] S. Rasp, M. S. Pritchard, P. Gentine, Deep learning to represent subgrid processes in climate models, Proceedings of the National Academy of Sciences 115 (39) (2018) 9684–9689.

[13] H. K. Kim, S. Min, M. Song, S. Jung, J. W. Choi, Y. Kim, S. Lee, S. Yoon, H. H. Kim, Deep learning improves prediction of crispr–cpf1 guide rna activity, Nature biotechnology 36 (3) (2018) 239.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional architecture for fast feature embedding, in: 22nd ACM Intl. Conf. on Multimedia, 2014, pp. 675–678.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., TensorFlow: A system for large-scale machine learning, in: OSDI, Vol. 16, 2016, pp. 265–283.

[16] F. Chollet, Deep Learning with Python, Manning Publications, 2017.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch,

in: NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques, 2017.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (Oct) (2011) 2825–2830.

[19] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, K. Chard, Serverless supercomputing: High performance function as a service for science (2019). arXiv:1908.04907.

[20] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, K. Chard, funcX: A federated function serving fabric for science, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, ACM, 2020. doi:10.1145/3369583.3392683.

[21] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, I. Foster, Dlhub: Model and data serving for science, in: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 283–292. doi:10.1109/IPDPS.2019.00038.

[22] R. Chard, L. Ward, Z. Li, Y. Babuji, A. Woodard, S. Tuecke, K. Chard, B. Blaiszik, I. Foster, Publishing and serving machine learning models with dlhub, in: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), PEARC '19, ACM, New York, NY, USA, 2019, pp. 73:1–73:7. doi:10.1145/3332186.3332246.

[23] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, J. Soyke, Tensorflow-Serving: Flexible, high-performance ML serving, arXiv preprint arXiv:1712.06139 (2017).

[24] O. E. Gundersen, S. Kjensmo, State of the art: Reproducibility in artificial intelligence, in: 30th AAAI Conf. on Artificial Intelligence, 2017, pp. 1644–1651.

[25] E. Gossett, C. Toher, C. Oses, O. Isayev, F. Legrain, F. Rose, E. Zurek, J. Carrete, N. Mingo, A. Tropsha, S. Curtarolo, AFLOW-ML: A RESTful API for machine-learning predictions of materials properties, Computational Materials Science 152 (2018) 134–145. arXiv:1711.10744, doi:10.1016/j.commatsci.2018.03.075.

[26] Q. Zhang, D. Chang, X. Zhai, W. Lu, OCPMDM: Online computation platform for materials data mining, Chemometrics and Intelligent Laboratory Systems 177 (November 2017) (2018) 26–34. doi:10.1016/j.chemolab.2018.04.004.

[27] A. Agrawal, A. Choudhary, An online tool for predicting fatigue strength of steel alloys based on ensemble data mining, International Journal of Fatigue 113 (2018) 389–400.

[28] M. Baker, 1,500 scientists lift the lid on reproducibility, Nature 533 (2016) 452–454. doi:10.1038/533452a.

[29] A. Morin, J. Urban, P. Adams, I. Foster, A. Sali, D. Baker, P. Sliz, Shining light into black boxes, Science 336 (6078) (2012) 159–160.

[30] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludscher, B. D. Mecum, J. Nabrzyski, V. Stodden, I. J. Taylor, M. J. Turk, K. Turner, Computing environments for reproducibility: Capturing the “Whole Tale”, Future Generation Computer Sys. (2018).

[31] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. Ioannidis, M. Taufer, Enhancing reproducibility for computational methods, Science 354 (6317) (2016) 1240–1241. doi:10.1126/science.aah6168.

[32] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, et al., XSEDE: Accelerating scientific discovery, Computing in Science & Engineering 16 (5) (2014) 62–74.

[33] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, et al., The Open Science Grid, in: Journal of Physics: Conference Series, Vol. 78, IOP Publishing, 2007, p. 012057.

[34] K. Chard, S. Tuecke, I. Foster, Globus: Recent enhancements and future plans, in: XSEDE16 Conference on Diversity, Big Data, and Science at Scale, ACM, 2016, p. 27.

[35] B. Blaiszik, K. Chard, J. Pruyne, R. Ananthakrishnan, S. Tuecke, I. Foster, The Materials Data Facility: Data services to advance materials science research, JOM 68 (8) (2016) 2045–2052.

[36] A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, et al., The Materials Project: A materials genome approach to accelerating materials innovation, APL Materials 1 (1) (2013) 011002.

[37] Caffe Model Zoo, `http://caffe.berkeleyvision.org/model_zoo.html`. Accessed November 8, 2019.

[38] ModelHub, `http://modelhub.ai/`. Accessed November 8, 2019.

[39] R. S. Olson, M. Sipper, W. La Cava, S. Tartarone, S. Vitale, W. Fu, P. Orzechowski, R. J. Urbanowicz, J. H. Holmes, J. H. Moore, A system for accessible artificial intelligence, in: Genetic Programming Theory and Practice XV, Springer, 2018, pp. 121–134.

[40] Algorithmia, `https://algorithmia.com/`. Accessed Novemeber 8, 2019.

[41] Kubeflow, `https://www.kubeflow.org/`. Accessed November 8, 2019.

[42] R. Ananthakrishnan, B. Blaiszik, K. Chard, R. Chard, B. McCollam, J. Pruyne, S. Rosen, S. Tuecke, I. Foster, Globus platform services for data publication, in: Practice and Experience on Advanced Research Computing, ACM, 2018, pp. 14:1–14:7.

[43] J. Forde, T. Head, C. Holdgraf, Y. Panda, G. Nalvarete, B. Ragan-Kelley, E. Sundell, Reproducible research environments with repo2docker, in: Reproducibility in Machine Learning Workshop, 2018.

[44] J. Starr, A. Gastl, Iscitedby: A metadata scheme for datacite, D-Lib Magazine 17 (1/2) (2011).

[45] Y. Babuji, K. Chard, I. Foster, D. S. Katz, M. Wilde, A. Woodard, J. Wozniak, Parsl: Scalable parallel scripting in Python, in: 10th International Workshop on Science Gateways, 2018.

[46] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, K. Chard, Parsl: Pervasive parallel programming in python, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '19, ACM, New York, NY, USA, 2019, pp. 25–36. `doi:10.1145/3307681.3325400`.

[47] R. Madduri, K. Chard, M. DArcy, S. C. Jung, A. Rodriguez, D. Sulakhe, E. Deutsch, C. Funk, B. Heavner, M. Richards, et al., Reproducible big data science: a case study in continuous fairness, PloS one 14 (4) (2019).

[48] S. Tuecke, R. Ananthakrishnan, K. Chard, M. Lidman, B. McCollam, S. Rosen, I. Foster, Globus Auth: A research identity and access management platform, in: 12th Intl Conf. on e-Science, 2016, pp. 203–212. `doi:10.1109/eScience.2016.7870901`.

[49] Theta, `https://www.alcf.anl.gov/theta`. Accessed November 8, 2019.

[50] Cooley, `https://www.alcf.anl.gov/resources-expertise/analytics-visualization`. Accessed November 8, 2019.

[51] Petrelkube, `https://press3.mcs.anl.gov/jlse/projects/petrelkube-jlse-kubernetes-testbed-pi-rick-wagner-dsl/`. Accessed November 8, 2019.

[52] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the Inception architecture for computer vision, in: IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2818–2826.

[53] A. Krizhevsky, Learning multiple layers of features from tiny images, Tech. rep., Citeseer (2009).

[54] S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, G. Ceder, Python Materials Genomics (pymatgen): A robust, open-source Python library for materials analysis, Computational Materials Science 68 (2013) 314–319.

[55] L. Ward, A. Dunn, A. Faghaninia, N. E. Zimmermann, S. Bajaj, Q. Wang, J. Montoya, J. Chen, K. Bystrom, M. Dylla, K. Chard, M. Astad, K. A. Persson, G. Jeffrey, Snyder, I. Foster, A. Jain, Matminer: An open source toolkit for materials data mining, Computational Materials Science 152 (2018) 60–69.

[56] S. Kirklin, J. E. Saal, B. Meredig, A. Thompson, J. W. Doak, M. Aykol, S. Rühl, C. Wolverton, The Open Quantum Materials Database (OQMD): Assessing the accuracy of DFT formation energies, npj Computational Materials 1 (2015) 15010. `doi:10.1038/npjcompumats.2015.10`.

[57] D. Michie, 'memo' functions and machine learning, Nature 218 (5136) (1968) 19.

[58] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. Van Essen, M. Baughman, CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research, in: Computational Approaches for Cancer Workshop, 2017.

[59] B. Blaiszik, L. Ward, M. Schwarting, J. Gaff, R. Chard, D. Pike, K. Chard, I. Foster, A data ecosystem to support machine learning in materials science, MRS Communications 9 (4) (2019) 1125–1133.