Developing Distributed High-performance Computing Capabilities of an Open Science Platform for Robust Epidemic Analysis

Nicholson Collier
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL, U.S.A.
https://orcid.org/0000-0002-2376-4156

Justin M. Wozniak
Data Science and Learning
Argonne National Laboratory
Lemont, IL, U.S.A.
https://orcid.org/0000-0002-2441-2048

Abby Stevens
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL, U.S.A.
https://orcid.org/0000-0003-1976-1806

Yadu Babuji
Computer Science
University of Chicago
Chicago, IL, U.S.A.
https://orcid.org/0000-0002-9162-6003

Mickaël Binois
Acumes project-team
Inria centre at Université Côte d'Azur
Sophia Antipolis, France
https://orcid.org/0000-0002-7225-1680

Arindam Fadikar
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL, U.S.A.
https://orcid.org/0000-0001-7396-0350

Alexandra Würth
Acumes project-team
Inria centre at Université Côte d'Azur
Sophia Antipolis, France
https://orcid.org/0000-0002-6099-0531

Kyle Chard
Computer Science
University of Chicago
Chicago, IL, U.S.A.
https://orcid.org/0000-0002-7370-4805

Jonathan Ozik
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL, U.S.A.
https://orcid.org/0000-0002-3495-6735

Abstract—COVID-19 had an unprecedented impact on scientific collaboration. The pandemic and its broad response from the scientific community has forged new relationships among domain experts, mathematical modelers, and scientific computing specialists. Computationally, however, it also revealed critical gaps in the ability of researchers to exploit advanced computing systems. These challenging areas include gaining access to scalable computing systems, porting models and workflows to new systems, sharing data of varying sizes, and producing results that can be reproduced and validated by others. Informed by our team's work in supporting public health decision makers during the COVID-19 pandemic and by the identified capability gaps in applying high-performance computing (HPC) to the modeling of complex social systems, we present the goals, requirements, and initial implementation of OSPREY, an open science platform for robust epidemic analysis. The prototype implementation demonstrates an integrated, algorithm-driven HPC workflow architecture, coordinating tasks across federated HPC resources, with robust, secure and automated access to each of the resources. We demonstrate scalable and fault-tolerant task execution, an asynchronous API to support fast time-to-solution algorithms, an inclusive, multi-language approach, and efficient wide-area data management. The example OSPREY code is made available on a public repository.

Index Terms—high-performance computing, computational epidemiology, HPC workflows, open science platform

I. INTRODUCTION

The societal importance of simulating social systems has been brought into sharper focus as a result of the COVID-19 pandemic. The pandemic has shown how epidemiologic modeling can inform decision-making in times of crisis and uncertainty, while also highlighting areas that must be addressed to create sustainable, efficient, and more effective approaches to understanding the relevant and complex social processes for biopreparedness and response. An area of particular interest has been in using high-performance computing (HPC) to calibrate and apply epidemiologic models for producing outputs such as forecasts of cases, resource needs, and disease outcomes, to gain insight into the evolving nature of the pandemic [1] and to provide quick turnaround decision support for public health stakeholders [2].

However, despite the unprecedented production [3], [4] and co-production of scientific work [5], [6], in the form of large ensemble forecasts and scenario modeling, individual research groups have generally worked independently, as they've sought to exploit [2] (or attempt to exploit) advances in HPC, data management, machine learning (ML), artificial intelligence (AI), and automation methods when developing, calibrating, modifying, verifying, and validating the epidemiologic models. This has necessitated a large amount of heroic, overlapping

work, with results that risk lacking robustness, security, scalability, or efficiency. Further, differences across HPC resources can result in one-off workflows, developed to meet the requirements of a specific HPC environment.

To add to the complexity, the data upon which modelers have relied is heterogeneous, changing, and incomplete, requiring complex integration across diverse and novel surveillance signals. These features present significant challenges for use in calibrating epidemiologic models. At the COVID-19 pandemic onset, data were generally limited to diagnosed cases, deaths, and sometimes hospitalizations favoring simpler models that did not require estimation of many parameters. As the pandemic evolved, multiple new high-resolution data streams emerged at the local level allowing the models to capture dynamic features of the epidemic and adapt towards better realism. Anticipating the evolution of data streams would help develop flexible models that could be quickly scaled up in complexity and ingest dynamically changing data that vary between locations; however, data management infrastructure is needed to efficiently support this.

Further, research groups have often focused on a single modeling scope, with one modeling method (compartmental, meta-population, agent-based), geographical extent (city, county, state, national), and temporal scale (short-term forecast, medium/long-term planning). Even when multiple scopes are considered, they are rarely integrated into multi-resolution ensembles that can mutually inform, and which could be combined to rapidly support decision making during different stages of an unfolding public health emergency.

To begin addressing these issues, we present the design and initial implementation of the Open Science Platform for Robust Epidemic analYsis, or OSPREY, which seeks to lower the barriers to and automate epidemiologic model analyses, monitoring, and rapid response on HPC resources. We begin by describing OSPREY's goals and requirements in section II, and then discuss related work in section III. Section IV presents our prototype OSPREY HPC architecture and section V describes the programming model and application programming interfaces (APIs) for working with the architecture components. Section VI provides results from an example optimization workflow built using the prototype architecture and APIs. We summarize our contributions and discuss future work directions in section VII.

II. OSPREY GOALS AND REQUIREMENTS

A. OSPREY Goals

Here we describe the three key OSPREY capabilities that drive its design.

1) Integrated, algorithm-driven HPC workflows: OSPREY needs to facilitate access to HPC resources for epidemiologic model developers and for developers of model exploration algorithms (e.g., calibration, data assimilation, uncertainty quantification, optimization algorithms). HPC workflow software will connect data, simulation, and algorithmic tasks, and provision heterogeneous resources (CPU, GPU, and other specialized accelerators) matched to task types, such as simulation-intensive (CPU) or machine learning model-intensive (GPU). Workflows driven by complex, iterating, and asynchronous algorithms must be executable at scales ranging from HPC clusters to the largest supercomputers [2], [7], while enabling modeling groups with different computing allocations and resource requirements to transparently utilize and build upon the complete suite of OSPREY capabilities by plugging code into an automated, scalable, and reusable framework.

2) Data ingestion, curation, and management: OSPREY will need to enable continuously running data assimilation analyses for melding data streams with up-to-date model forecasts. The platform will ingest real-time and near real-time data streams; curate, store, and index data; and manage epidemiologic models and model outputs. Data pipelines will quantify and adjust for data limitations and track data provenance.

3) Shared Development Environment (SDE) for rapid response and collaboration: The OSPREY design is a response to our experiences in utilizing HPC resources in support of public health decision-making during the COVID-19 pandemic. Computational experimentation, verification, and validation are collaborative activities that require rapid cooperative development as public health crises evolve. Quickly and correctly porting various modeling and analysis codes to HPC resources is also an important activity that requires a range of expertise. Thus, the OSPREY design will make available shared, flexible, automated, and scalable approaches to support rapid model exploration in a Shared Development Environment (SDE).

B. OSPREY Requirements

Next we describe the requirements needed for OSPREY to address the goals outlined above. First we present the main focus for this paper: requirements for integrated, algorithm-driven HPC workflows. The final two sections (II-B2,II-B3) describe other requirements related to "Data ingestion, curation, and management" and "Shared Development Environment (SDE) for rapid response and collaboration" that are not the focus of this work, but we include for completeness.

1) Integrated, algorithm-driven HPC workflows:

a) Coordinated multi-resource task execution: The breadth of model, simulation, and analyses employed in OSPREY necessitates a flexible approach to computation in which a range of computational tasks can be executed across a distributed ecosystem of heterogeneous remote resources. Such tasks are varied and may include single-core tasks to multi-node MPI tasks, running on CPUs to AI accelerators, and requiring different performance guarantees (e.g., response time) to coordinate execution of complex workflows.

b) Robust, secure, and automated access to federated HPC resources: HPC and supercomputing resources have a wide range of access, authentication, and security protocols. OSPREY will need to allow for robust, secure, and automated access to computational resources.

c) Scalable, fault-tolerant task execution: Computational demand in epidemiologic workflows can vary dramatically

over time. Furthermore, computational availability can fluctuate due to demand, resource priorities, and site specific preemption protocols. Being able to dynamically scale up (or down) the resources available for task execution, while retaining the integrity of the overall workflows is critical.

d) Fast time-to-solution workflows: Epidemiologic models have become essential tools for understanding and characterizing disease evolution. However, for them to be useful as decision support tools, they need to provide actionable insights quickly. There is, therefore, a need to develop and apply algorithms focused on fast time-to-solution, whether for model calibration, optimization, or other exploration. These include efficient, surrogate-based multi-objective optimization algorithms that can exploit the concurrency of HPC resources [2], [8], and asynchronous algorithms that can incrementally learn and adjust as new information is obtained. Being able to leverage the latest advances in data analytic and statistical libraries is desirable. OSPREY needs to provide the ability to effectively integrate and exploit such algorithms.

e) Multi-language workflows: Computational epidemiology is inherently cross disciplinary, bringing together a wide array of expertise from many domains. There is, therefore, not a single *lingua franca* that can be assumed for developing the model exploration algorithms that drive the workflows. OSPREY will need to be inclusive and provide multi-language APIs for workflows.

f) Efficient wide-area data staging: Data is fundamental to epidemiologic analyses, whether in the form of traditional data structures or ML/AI/mechanistic model artifacts. Workflow computations require efficient staging of these data elements and need to provide uniform access to them across the heterogeneous computing infrastructure.

2) Data ingestion, curation, and management: OSPREY will need to provide reusable services to make it possible to quickly develop and deploy workflows based on real-world data streams and model artifacts on HPC resources.

a) Data stream ingestion: Incoming data streams relevant to OSPREY workflows vary widely in type and size. OSPREY will need to develop flexible techniques to move and track data sets from their origin of publication, such as a city or health department portals, to their site of use, such as a HPC cluster or supercomputer, to the models they are used in.

b) Automated data curation: To address the heterogeneous, changing, and incomplete data typical of surveillance and other public health data that OSPREY workflows will utilize, there is a need to develop capabilities for creating data analysis pipelines, such as for data de-biasing, data integration, uncertainty quantification, and more general metadata and provenance tracking.

c) Managing algorithm and model artifacts: Algorithm and model artifacts, such as model exploration state or calibrated model checkpoints, can be complex, large, and numerous and not local to a specific resource. OSPREY needs to manage these artifacts, and their associated metadata. Capabilities should allow model exploration algorithms to

be easily rerun or continued, either on the original set of computing resources or different ones. Model checkpoints should be easily selected, staged for execution, and run.

3) SDE for rapid response and collaboration: OSPREY will make available a Shared Development Environment (SDE) to make it possible to quickly share, validate, and scale models and workflows on HPC resources. A notable feature of the OSPREY SDE is that it is not based on hardware or Infrastructure-As-A-Service (IaaS) products, but rather on portable workflows that run on the federated HPC systems (§II-B1b) to which the user already has access.

a) Model and workflow sharing: Sharing scientific workflows is known to be difficult due to the typical intricacies of interactions with complex HPC systems components including shared filesystems, schedulers, and various environmental settings. The standardized OSPREY workflow structure and its use of portable tools will make it more likely that "works for me" also means it will "work for you," at least at the systems level.

b) Model validation and publishing: At the scientific level, models must be calibrated, validated, and published using well-defined data sets. We will employ best practices from the DevOps ecosystem [9] to make it easier for modelers to post complete models with the data used to validate them for reproduction, extension, or scaling by others, with the capability to detect correctness regressions.

III. RELATED WORK

The desire to perform secure remote computation, particularly for scientific computations, is not new, with methods such as SSH commonly used to securely execute commands over insecure network connections. Recent paradigms, such as Grid computing and cloud computing, have introduced more seamless methods for remote computing via common APIs, authentication frameworks, and infrastructure abstractions.

The Globus Grid Resource Allocation Manager (GRAM) [10] defined a common, web-accessible API for interacting with various batch schedulers. Computing facilities, such as NERSC and TACC, are developing REST APIs to, amongst other things, submit and manage batch jobs (e.g., SuperFacility [11] and Tapis [12]).

Science gateways [13] provide web-based interfaces to various science tools, resources, and data. They have become popular as they abstract the complexity of dealing with remote environments, consolidate the tools and resources needed for a specific application, and remove the need to use several interfaces. Gateways are typically developed to support a specific domain or application; however, many build upon general-purpose frameworks, such as Apache Airavata [14], Tapis, and Globus [15] that provide a range of data and compute management services.

Workflow management systems, such as Pegasus [16], Parsl [17], Swift [18], and Balsam [19], are designed to simplify the development and execution of sophisticated workflows—tasks defined and specified to be performed in

a partial order with concurrent semantics. Workflow management systems enable workflows to be executed on various computing resources including clouds, grids, and HPC clusters. These systems typically are deployed locally on a compute resource, and use pilot jobs to manage execution of many tasks within a batch scheduler job. Some workflow management systems support multi-site execution; however, this typically relies on establishing (and maintaining) SSH connections or opening ports to shared databases or controllers.

There is growing interest in remote computation as part of AI-based workflows. For example, Colmena [20] is a Python-based framework designed to steer computational campaigns by enabling developers to wrap various fidelity tasks (e.g., simulations) and define functions to select which tasks to be executed next. Colmena supports the use of both Parsl and funcX [21] (§IV-B) to manage execution of tasks on HPC resources.

IV. PROTOTYPE OSPREY HPC ARCHITECTURE

In this section, we describe the prototype OSPREY HPC architecture, which we generalize from the EMEWS framework [22]. The architecture is depicted in Figure 1. The architecture consists of the algorithm that controls the overall workflow (§IV-A), a control and task distribution system (§IV-B), a distributed task execution service (§IV-C), worker pools (§IV-D), and a data sharing service (§IV-E).
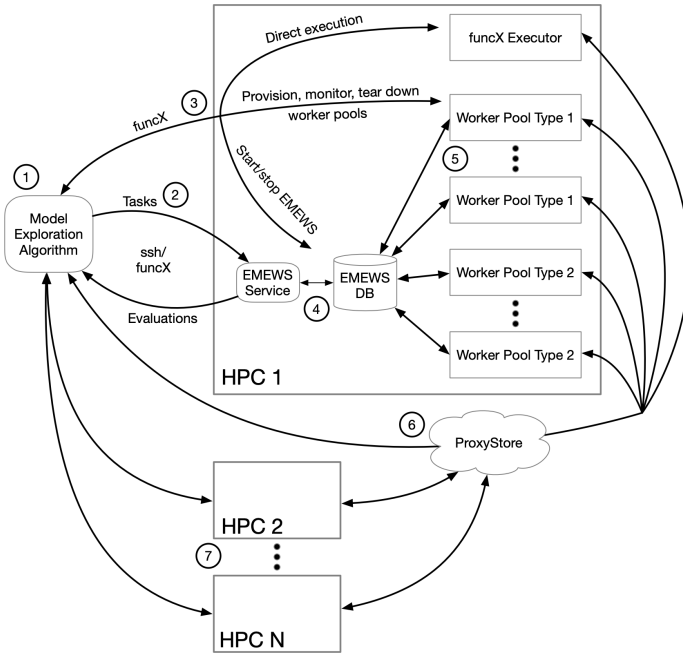

Fig. 1. Overview diagram of OSPREY prototype functionality.

A. The Model Exploration Algorithm Module

The central control of the OSPREY workflow is in the logic of the Model Exploration (ME) algorithm ①. This is the main user interface to the system. The users deploy scientifically-oriented algorithms here, using a wide range of supported languages, including Python and R. This allows users to draw from the extensive set of community-developed data science and analysis tools available. Such algorithms typical provide a pluggable interface in which to provide user functions to sample the scientific problem space. This is where the user connects the algorithm to the OSPREY task API.

The API for task submission, result reporting, and querying the queues is implemented in both Python and R, although the Python API has additional code that enables asynchronous ME algorithms (§V-B). A task is submitted with the following arguments: an experiment id; the task work type; the task payload; an optional priority that defaults to 0; and an optional metadata tag string. The payload contains sufficient information for a worker pool to execute the task and is typically a JSON formatted string, either a JSON dictionary or in less complex cases a simple JSON list. On submission, the API creates a unique task identifier (an integer) for the task and inserts that identifier, the experiment identifier, the work type, and the payload into the EMEWS DB (§IV-C) tasks table, together with a task creation timestamp. That task identifier, priority and work type are then inserted into the EMEWS DB output queue table.

B. Task Distribution

Tasks produced by the ME algorithm are distributed over the wide area network via a configurable network, with funcX [21] or SSH as the transport mechanism ②. Overall, the task distribution system decouples the tasks produced by the ME algorithm, and the status of those tasks (queued, running, etc.) from the ME execution such that tasks and their results are not lost when a resource fails, but rather are described in the system in enough detail so that they can be executed if not yet running or restarted if necessary.

We have built the OSPREY computational fabric upon the Argonne-developed funcX platform [21]. FuncX implements a federated function as a service (FaaS) model via which arbitrary Python functions can be reliably executed on remote computers. Users first deploy specialized funcX endpoint software on a computer to make it accessible for remote computation. The hosted funcX cloud service acts as an interface for users to submit tasks. The service is responsible for managing secure communication with an endpoint, authenticating and authorizing users (via OAuth 2.0), providing fire-and-forget execution by storing and retrying tasks in the event an endpoint is offline or fails, and storing results (or failures) until retrieved by a user. The funcX endpoint is responsible for provisioning resources via various supported systems (e.g., local fork, Slurm, PBS), managing execution of tasks using a pilot job model, optionally deploying tasks within containers, monitoring execution, and returning task results to users via the cloud service.

In our prototype, we use funcX to start and stop the EMEWS service, the EMEWS DB database (§IV-C), and remote worker pools on HPC resources (§IV-D) ③. The

EMEWS service is a Python application and can thus be started directly from within a Python function executed on a remote funcX endpoint. The database and worker pools are non-Python applications, and those are started using funcX and the Python subprocess library to start the relevant executables. Due to the flexibility of funcX, we can also use it to execute simple, standalone tasks on remote resources as need, such as for initiating large data transfers, or, as we show in §VI, for an optimization phase in a Python ME algorithm that is best executed on a specific HPC resource.

C. EMEWS Task Database

Tasks arrive at HPC sites at the EMEWS Service ④, which abstracts task caching and queuing operations in an efficient manner. The Service stores the tasks in the EMEWS DB, a resource-local SQL database. The Service mediates between model exploration algorithms and worker pools and exposes data about tasks for queries.

The EMEWS DB consists of five tables, a tasks table, a table each for an input and output queue, an experiments table that links tasks with experiments, and a tag table that links individual tasks to a metadata tag or tags. The tasks table contains the data for each task: a unique task identifier; current status (queued, running, complete, or canceled); a task work type; the input payload (e.g., simulation input parameters); the result payload; the identifier of the worker pool running the task; the time the task was created; the task execution start time; and the task execution stop time. The output queue table, from which tasks are popped for execution, consists of a task identifier, a task work type, and a priority that determines the order in which the tasks should be executed. The input queue table consists of a task identifier and a task type. The rows in the tables are linked through the task identifier such that tasks in the input and output queue and in other tables link to those in the tasks table via the shared task identifier. This schema provides the foundation for a fault tolerant and robust task queuing and execution workflow.

Client code can query for tasks in the output queue by passing an integer specifying the type of work, the number of tasks to retrieve, an optional string identifying the specific worker pool consuming the tasks, and an optional timeout and delay value. The query executes by polling the output queue table for tasks of the specified work type, using the specified delay and timeout value. On success, the highest priority task (or tasks if multiple tasks are requested) is deleted and returned from the output table, effectively popping the task off the output queue. The task's payload is retrieved from the task table which is updated, setting the task status to running, recording the start time, and setting the worker pool identifier to that passed in by the query. The payload and the task identifier are returned as a Python dictionary or a R named list: `{'type': 'work', 'eq_task_id': task_id, 'payload': payload}`. If the polling fails, a dictionary or named list with a `'type'` of `'status'` and a `'payload'` describing the reason for the failure (e.g., `'TIMEOUT'`) is returned.

When the work defined by the `'payload'` has been completed, the result, typically in JSON format, is reported and inserted into the input queue by passing the task identifier, the work type, and the result JSON payload. The result payload is inserted into the tasks table where the task status is marked as completed, and the stop time is recorded. The task identifier and work type are inserted into the input queue table, effectively pushing the task into the input queue. Client code, such as an ME algorithm, queries for results by polling the input queue table for completed tasks, passing a task identifier together with timeout and delay values that work identically to those in the output queue query. On success, the task is popped off the input queue by deleting it from the input queue table, and the corresponding result from the tasks table is returned. On failure, an appropriate message is returned.

D. Heterogeneous Worker Pools

In the previous section (§IV-B) we described how tasks can be submitted to, and how results are retrieved from the EMEWS DB. Worker pools ⑤ are responsible for querying the database via the output queue for those submitted tasks, executing them, and then reporting the results of those tasks back to the database where the ME algorithm can then retrieve them via the input queue. Our canonical worker pool implementation is a Swift/T [23] application written using the Swift language and Python, running on a HPC resource as a pilot job. The ability to manage an extreme quantity of tasks is a main design feature of the Swift/T implementation, which essentially distributes work among previously launched workers using MPI messages. Swift/T is a dataflow language with built-in concurrency designed for execution on large-scale supercomputers. Our worker pool implementation leverages these features to execute as many submitted tasks in parallel as possible. For performance results of epidemiologic models running on Swift/T workflows see [2].

Swift/T also provides high-level, easy to use interfaces for Python, R, Julia and Tcl, allowing the user to pass a string of code into the language interpreter for execution. This feature enables us to integrate Swift's concurrent capability with the Python EMEWS DB API for querying the output queue for tasks to execute, and for pushing the results of executed tasks on to the input queue. We have also implemented an enhanced version for querying the output queue, customized for worker pools. These queries allow a worker pool to request up to n number of tasks (a query *batch size*) to consume at a time, while accounting for the number of tasks a worker pool already has obtained but have not completed. So, for example, if a worker pool is configured to possess 33 tasks at a time, if it owns 30 uncompleted tasks when querying the output queue, it will only obtain 3 additional tasks. This can be tweaked using a *threshold* value that specifies how large the deficit between requested tasks and owned tasks must be before more tasks are obtained. Querying for tasks in this way allows a worker pool to tune its query to the number of available workers such that all its workers are busy while equitably sharing work among multiple worker pools. This prevents any one worker

pool from obtaining more tasks than it can reasonably execute while potentially leaving other pools starved of work.

Each worker pool has a user specified work type associated with it when it is initialized. A pool will only query for and execute tasks of that type. Consequently, worker pools can be matched to HPC resources configured to most efficiently execute their work type. For example, an ME algorithm may have two types of tasks that need to be executed: 1) a multi-process MPI-based simulation model; and 2) an optimization component that most efficiently runs on a GPU. Two worker pools can be launched and configured on resources appropriate for these two different work types. Worker pools also need not be started and stopped when the ME algorithm starts and stops. They can be started and stopped as needed in response to changing ME algorithm requirements.

Worker pools can run a variety of task application types. In addition to the Python, R, Julia, and Tcl interfaces which can be used to run code written in those languages, Swift/T can run executables as command line programs via its `app` function type. MPI executables can also be run via an `app` function or internally as parallel Swift/T library extensions using the `@par` keyword.

E. Data Sharing Service

One of the challenges with our approach is efficiently moving data to/from the HPC resources. Staging data (including models) via, e.g., SCP requires authentication (often two-factor authentication on HPC resources), while using funcX is not possible as funcX limits input/output sizes to 10MB. To address the need for out-of-band transfer of potentially large data, we use ProxyStore [24] and Globus [15]. This communication is shown in Figure 1 ⑥ and allows for seamless access to remotely-hosted large data sets across HPC sites, with no changes to code.

ProxyStore is a data management fabric that exposes a simple (and common) Python interface to data irrespective of where it resides. Specifically, it passes "Proxy" object references between participating entities (e.g., ME algorithms, EMEWS DB, remote workers) and implements a lazy evaluation approach in which Proxies are resolved only when needed. Thus, users are presented with a pure Python interface which can be easily integrated with various models and client software. ProxyStore implements a common data access/movement interface with plugins to support storage and movement via different methods, including shared file systems, Redis databases, or Globus.

Globus provides high-performance and reliable third-party data transfer and is available on HPC systems in national laboratories, national cyberinfrastructure, and in research institutions. It can also be easily deployed on laptops and clouds. The third-party nature of Globus transfers, allows OSPREY (via ProxyStore) to easily move data between locations without needing to maintain open connections to those locations.

The capabilities of the OSPREY prototype architecture described, including the ME algorithm (§IV-A), the control

and task distribution system (§IV-B), the distributed task execution service (§IV-C), the worker pools (§IV-D), and the data sharing service (§IV-E), provide the ability to robustly and securely coordinate workflows across a distributed ecosystem of heterogeneous remote resources ⑦.

V. PROGRAMMING MODEL AND APIS

A. Task model

Examples of the base API for submitting tasks, querying for tasks to execute, reporting results, and querying for results in both Python and R can been seen in Listing 1. The Python version currently has some additional functionality, the ability to query for multiple tasks to execute, for example, which accounts for differences in the method and function signatures. Further, the Python API is class based in which instances of an `EQSQL` Python class provides methods for submission, querying and reporting.

```
# Python API
def submit_task(self, exp_id: str,
    eq_type: int, payload: str,
    priority: int = 0,
    tag: str = None)

def query_task(self, eq_type: int, n: int = 1,
    worker_pool: str = 'default',
    delay: float = 0.5,
    timeout: float = 2.0)

def report_task(self, eq_task_id: int,
    eq_type: int, result: str)

def query_result(self, eq_task_id: int,
    delay: float = 0.5,
    timeout: float = 2.0)

# R API
eq_submit_task <- function(exp_id, eq_type,
    payload, priority=0)

eq_query_task <- function(eq_type, delay = 0.5,
    timeout=2.0)

eq_report_task <- function(eq_task_id,
    eq_type, result)

eq_query_result <- function(eq_task_id,
    delay = 0.5, timeout = 2.0)
```

Listing 1. Core EMEWS DB API in Python and R.

B. Asynchronous Tasks

In section II-B1d, we discussed the importance of asynchronous algorithms for fast time to solution, and for providing better utilization of HPC resources when compared with batch synchronous workflows. Here we describe the OSPREY asynchronous API that enables these algorithms. The asynchronous API is designed using the *future* abstraction. A future encapsulates the asynchronous execution of a task, and is implemented as a Python class. Future instances are created and returned when tasks are submitted with `EQSQL.submit_task`. Leveraging the EMEWS DB API, `Future` class methods allow ME algorithm code to query

```
1: for each initial sample do
2:     Submit the sample for evaluation
3: end for
4: while stopping condition not reached do
5:     Wait for n number of evaluation results
6:     Re-sample, reorder, re-submit based on results
7: end while
```

Fig. 2. Pseudo-code for an asynchronous algorithm.

the status (running, finished, etc.) and check for a result of the encapsulated task without waiting for it to finish. Other methods provide the ability to cancel and reprioritize the task with respect to other tasks in the output queue.

The asynchronous API also includes functions for working with collections of `Futures`. Given a list of `Futures`, the `as_completed` function creates a Python generator that will yield a specified number of `Futures` as they complete. `pop_completed` returns the first completed `Future` from a list of `Futures`, removing that `Future` from the list, and `update_priority` will update the priorities of a list of `Futures` to a new set of priorities. For efficiency, these functions typically perform batch operations on the EWEWS DB rather than iterating through the collection of `Futures` and performing the operations individually. Taken together, these functions and the `Future` class methods enable the implementation of various asynchronous algorithms.

The pseudo-code in Figure 2 illustrates a typical asynchronous algorithm. A number of initial samples are submitted for evaluation. After some number of evaluations have completed and their results are available, the ME algorithm, using these results, can generate new samples for evaluation, reorder existing evaluations, cancel less promising evaluations, and so on, until some stopping condition is reached. The code in Listing 2 is a possible implementation of the pseudo-code using our asynchronous API. Lines 4 through 9 create JSON payloads for each sample in a list of samples, submitting those for evaluation using the `EQSQL` class instance created on line 2. The `Futures` returned by the submission are added to a `futures` list in line 9. In line 13, the `pop_completed` function pops a completed future from the list. In this case, and by default, `pop_completed` will poll for a completed result until one is found. However, a maximum wait time can also be specified using a `timeout` argument. In line 15, an `update` function takes the result of the completed future, and generates new tasks to be submitted, and new priorities for the existing tasks. Line 16 updates the existing tasks encapsulated by the `futures` list to the new priorities, and lines 17-20 submit the new tasks, producing new futures, which are then added to the `futures` list where they can be popped off as they complete.

```
1 from eqsql import eq
2 eqsql_db = init_esql(...)
3 futures = []
```

```
4  for s in samples:
5      payload = json.dumps({'sample': sample})
6      ft = eqsql_db.submit_task(
7          'exp1', sim_work_type, payload,
8          priority=0)
9      futures.append(ft)
10
11 tasks_completed = 0
12 while tasks_completed < 1000:
13     ft = eq.pop_completed(futures)
14     tasks_completed += 1
15     tasks, new_priority = update(ft.result())
16     eq.update_priority(futures, new_priority)
17     for t in tasks:
18         ft = eqsql_db.submit_task(
19             'exp1', sim_work_type, task)
20         futures.append(ft)
```

Listing 2. Sample asynchronous algorithm implementation.

VI. RESULTS

To exercise our prototype OSPREY HPC architecture, we have implemented an example optimization workflow that attempts to find the minimum of the Ackley function [25] using a Gaussian process regression model (GPR). Our implementation, which can be found at https://github.com/NSF-RESUME/2023_ParSocial_OSPREY_example, is based on a similar example problem provided as part of the Colmena documentation [26]. We begin with a sample set containing a number of randomly generated n-dimensional points. Each of these points is submitted as a task to the Ackley function for evaluation. When a specified number of tasks have completed (i.e., that number of Ackley function evaluation results are available), we train a GPR using the results, and reorder the evaluation of the remaining tasks, increasing the priority of those more likely to find an optimal result according to the GPR. This repeats until all the evaluations complete.

Our example workflow was executed locally on an M1 Mac-Book Pro in conjunction with the University of Chicago's Midway2 HPC cluster, the Laboratory Computing Resource Center's Bebop HPC cluster at Argonne National Laboratory, and the Argonne Leadership Computing Facility (ALCF) Theta supercomputer. The EMEWS DB components and worker pools were run on Bebop, and the GPR training was done on Midway2 or Theta depending on the run configuration.

The ME algorithm is a Python script running locally that begins by initializing a funcX client, and then starting the EMEWS DB, an initial worker pool, and the EMEWS service remotely on Bebop using funcX as described in section IV-B. After initializing an SSH tunnel through which we communicate with the EMEWS service, we create an initial sample set of 750 4-dimensional points, which are submitted as tasks using the `submit_task` API function. The worker pool, running on Bebop, pops these tasks off the database output queue, and executes the Ackley function using the point data in the tasks' payload. (We have added a lognormally distributed "sleep" delay to the Ackley function implementation to increase the otherwise millisecond runtime and to add task runtime heterogeneity for demonstration purposes.) On completion, the task results are pushed onto the database's input queue. While the worker pool on Bebop is executing

the tasks, the local Python script, waits for the next 50 tasks to complete at which time we perform the reprioritization. The completed tasks are popped off the list of futures returned by the submission using the `as_completed` API function.

The reprioritization consists of retraining the GPR with the completed results and then updating the evaluation priorities of the uncompleted tasks using the GPR predictions. The retraining of the GPR was performed on Midway2 or Theta using funcX to directly evaluate the Python function that retrains the GPR and returns the updated evaluation order. The GPR itself was passed as a ProxyStore proxy object, using ProxyStore's Globus functionality, to the reprioritization function and resolved into the actual GPR during remote function evaluation. Using the updated order returned from the function, the uncompleted tasks are reprioritized using the `update_priorities` API function. The reprioritization repeats for every new 50 completed tasks, and start an additional worker pool on Bebop after the 2nd and 4th reprioritizations, for a final total of 3 worker pools. Connecting to the same database as the initial worker pool, these worker pools perform the same type of work, popping tasks off the same output queue, and executing the Ackley function using those tasks' payload. When there are no more tasks left to complete, the workflow terminates, stopping the database, and shutting down funcX.

To illustrate the workflow's execution, we have created two figures. The first, Figure 3, illustrates the effect of query batch size and threshold on worker pool utilization when querying for tasks. As mentioned in section IV-D batch size controls the maximum number of tasks a worker pool can own and the threshold determines when additional tasks are requested. The top plot in the figure shows the number of concurrently running tasks with a query batch size of 50 and a threshold of 1 for a worker pool with 33 workers. The middle plot shows a batch size of 33 with a threshold of 1 for the same size worker pool, and the bottom plot a batch size of 33 with a threshold 15, again for the same size worker pool. The top plot clearly shows the best utilization of the HPC resource, in this case, a single 36 core compute node on Bebop. A batch size of 50 with 33 workers oversubscribes the pool, and effectively creates an in-memory task cache from which new tasks can be quickly pulled without the more costly database query. Oversubscribing, however, consumes database tasks, making them ineligible for reprioritization or cancellation, since they are popped off the output queue. With a batch size of 33 and threshold of 1 (middle plot), there is no such cache, and each time a task is completed another must be fetched from the database, during which additional tasks may complete, leading to lower utilization, but making more tasks eligible for reprioritization or cancellation. The final plot illustrates the effect of a large threshold where 15 tasks must finish before new tasks are added resulting in the saw tooth pattern where multiple workers remain idle for several seconds at a time.

In Figure 4 we have plotted the number of tasks executed by each of the three worker pools over time, together with the GPR reprioritization. The bottom of the figure shows the
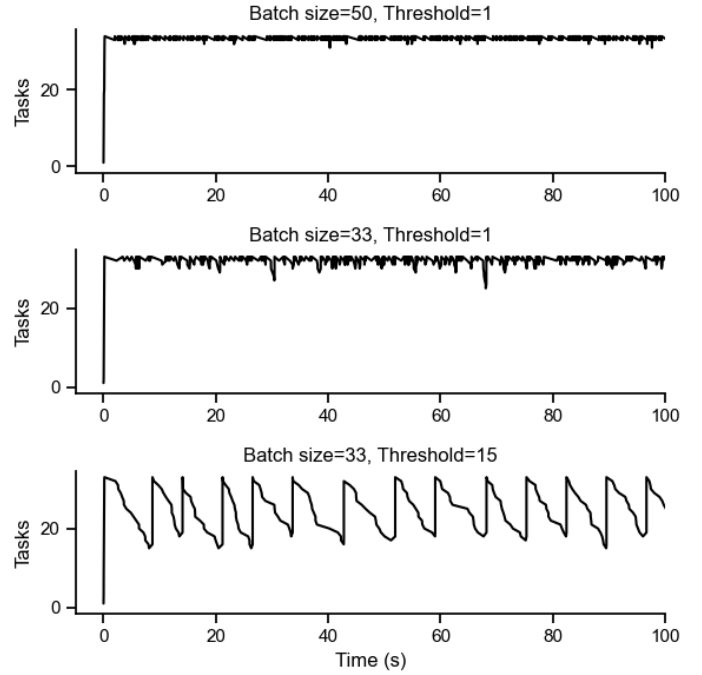

Fig. 3. Number of tasks executed by a worker pool for different batch sizes and thresholds.

total number of concurrently executing tasks by worker pool, each of which has been allocated 33 workers and thus can execute up to that many tasks concurrently. In this example, each worker pool has a batch size of 33 and thus will query the database for up to 33 tasks at a time, and request more when the number of owned tasks falls below 33 (threshold of 1). In the figure, we see worker pool 1, in blue, starting at time 0 and begin executing 33 concurrent tasks, indicated by the first dotted line. When a task finishes, we see the total number of tasks drop below the maximum of 33 and then rise back up as more tasks are requested and executed. 57 seconds after worker pool 1 has started, worker pool 2, in orange, starts, and executes its maximum of 33 concurrent tasks. At the 80 second mark, worker pool 3 starts and begins to evaluate tasks. Across all the worker pools, we see a similar utilization to that in the middle plot in Figure 3. By using the batch API, we can equitably spread the tasks among the pools.

The top of the figure illustrates the dynamics of the GPR reprioritization occurring while the worker pools consume tasks. The intermittent horizontal lines indicates the starting time and duration of the reprioritization with respect to worker pool task execution. We can see here how reprioritization becomes more frequent as the additional worker pools are added, given that 50 additional tasks complete more frequently with the additional worker pools. The very top of the figure shows the reprioritization trajectories of each of the 750 tasks, each line being drawn from a task's current priority to its new priority at the time of reprioritization. During the first reprioritization starting at the 29 second mark, and triggered by the first 50 tasks completing, 700 uncompleted

tasks are reprioritized with new priorities of $1 - 700$, at the next reprioritization 650 uncompleted tasks are reprioritized from $1 - 650$, and so on. During reprioritization, the worker pools continue to consume tasks, efficiently using the compute resources, and providing the GPR with additional results to evaluate. Also of note, is that although worker pool 2 and worker pool 3 are scheduled to start during the 2nd and 4th reprioritizations, respectively they do not immediately start consuming tasks at that time due to delays between submitting a worker pool job to Bebop and it actually beginning.
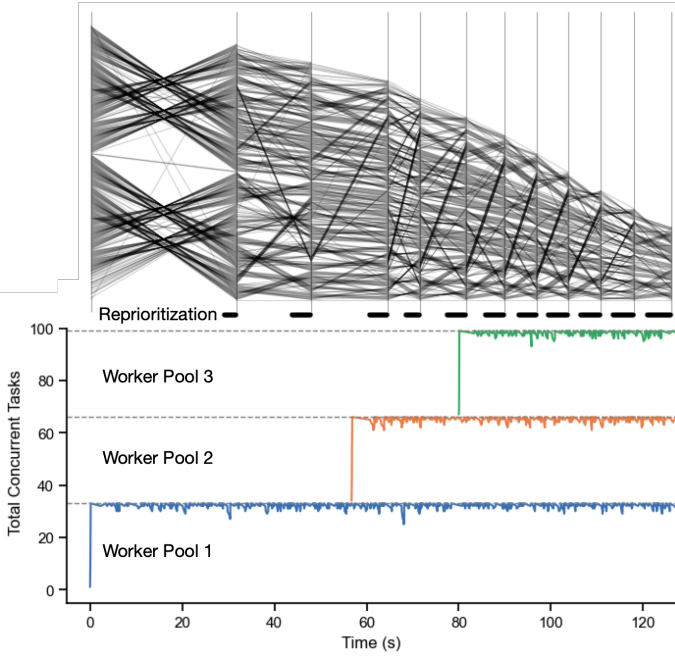

Fig. 4. Illustration of the combined example workflow across the ALCF Theta and LCRC Bebop resources. Top: GPR reprioritization of tasks over time, run on ALCF Theta. Vertical lines represent the new prioritization at the end of the reprioritization process, and connecting lines show task priority reassignments (higher means higher priority). As tasks are consumed, the tasks that are subject to reprioritization are reduced. The horizontal lines below the vertical lines show the time extent of each reprioritization call. Bottom: Total number of concurrently executing tasks by worker pool, run on LCRC Bebop.

VII. CONCLUSION AND FUTURE WORK

We have provided the goals and requirements for OS-PREY, an open science platform for robust epidemic analysis. We described the prototype HPC OSPREY architecture and demonstrated an example of its use across federated HPC resources, providing the implementation code on a public repository. Our goal is to develop OSPREY into a public resource for epidemiologic modeling and analysis.

There are multiple directions for future work. To meet additional requirements in the integrated, algorithm-driven HPC workflow topic (see II-B1), we will extend the asynchronous API to additional ME algorithm languages, starting with R, and expand the funcX capabilities for more robust interactions with HPC schedulers, including active monitoring and termination of worker pools, through the PSI/J library [27].

For OSPREY capabilities in data ingestion, curation, and management (II-B2), we will develop flexible techniques to provide real time and near-real time data to HPC workflows, automate the curation of the data, and provide methods for managing algorithm and model artifacts. Finally, to support the diffusion of OSPREY capabilities, we will develop the OSPREY shared development environment (II-B3), to promote model and workflow sharing, and to support best practices for model validation and reproducibility. These capabilities will allow us to expand the applicability of OSPREY to diverse ME algorithms and integrated HPC workflows, with the purpose of creating on-demand response and planning capabilities to support public health decision making.

Finally, while the focus of this work is in the public health domain, many aspects are relevant to other application areas where dynamic data and modeling on HPC resources can inform time-critical decision making.

REFERENCES

[1] A. L. Hotton, J. Ozik, C. Kaligotla, N. Collier, A. Stevens, A. S. Khanna, M. M. MacDonell, C. Wang, D. J. LePoire, Y.-S. Chang, I. J. Martinez-Moyano, B. Mucenic, H. A. Pollack, J. A. Schneider, and C. Macal, "Impact of changes in protective behaviors and out-of-household activities by age on COVID-19 transmission and hospitalization in Chicago, Illinois," *Annals of Epidemiology*, p. S1047279722001053, 6 2022, [Online; accessed 2022-06-23]. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1047279722001053

[2] J. Ozik, J. M. Wozniak, N. Collier, C. M. Macal, and M. Binois, "A population data-driven workflow for COVID-19 modeling and learning," *The International Journal of High Performance Computing Applications*, vol. 35, no. 5, pp. 483–499, 9 2021, [Online; accessed 2021-09-27]. [Online]. Available: http://journals.sagepub.com/doi/10.1177/10943420211035164

[3] H. Else, "How a torrent of COVID science changed research publishing — in seven charts," *Nature*, vol. 588, no. 7839, pp. 553–553, 12 2020, [Online; accessed 2021-09-28]. [Online]. Available: http://www.nature.com/articles/d41586-020-03564-y

[4] X. Cai, C. V. Fry, and C. S. Wagner, "International collaboration during the COVID-19 crisis: Autumn 2020 developments," *Scientometrics*, vol. 126, no. 4, pp. 3683–3692, 4 2021, [Online; accessed 2021-09-28]. [Online]. Available: https://link.springer.com/10.1007/s11192-021-03873-7

[5] E. L. Ray, N. Wattanachit, J. Niemi, A. H. Kanji, K. House, E. Y. Cramer, J. Bracher, A. Zheng, T. K. Yamana, X. Xiong, S. Woody, Y. Wang, L. Wang, R. L. Walraven, V. Tomar, K. Sherratt, D. Sheldon, R. C. Reiner, B. A. Prakash, D. Osthus, M. L. Li, E. C. Lee, U. Koyluoglu, P. Keskinocak, Y. Gu, Q. Gu, G. E. George, G. España, S. Corsetti, J. Chhatwal, S. Cavany, H. Biegel, M. Ben-Nun, J. Walker, R. Slayton, V. Lopez, M. Biggerstaff, M. A. Johansson, and N. G. Reich, "Ensemble forecasts of coronavirus disease 2019 (COVID-19) in the U.S." [Online]. Available: http://medrxiv.org/lookup/doi/10.1101/2020.08.19.20177493

[6] R. K. Borchering, C. Viboud, E. Howerton, C. P. Smith, S. Truelove, M. C. Runge, N. G. Reich, L. Contamin, J. Levander, J. Salerno, W. van Panhuis, M. Kinsey, K. Tallaksen, R. F. Obrecht, L. Asher, C. Costello, M. Kelbaugh, S. Wilson, L. Shin, M. E. Gallagher, L. C. Mullany, K. Rainwater-Lovett, J. C. Lemaitre, J. Dent, K. H. Grantz, J. Kaminsky, S. A. Lauer, E. C. Lee, H. R. Meredith,

J. Perez-Saez, L. T. Keegan, D. Karlen, M. Chinazzi, J. T. Davis, K. Mu, X. Xiong, A. Pastore y Piontti, A. Vespignani, A. Srivastava, P. Porebski, S. Venkatramanan, A. Adiga, B. Lewis, B. Klahn, J. Outten, J. Schlitt, P. Corbett, P. A. Telionis, L. Wang, A. S. Peddireddy, B. Hurt, J. Chen, A. Vullikanti, M. Marathe, J. M. Healy, R. B. Slayton, M. Biggerstaff, M. A. Johansson, K. Shea, and J. Lessler, "Modeling of future COVID-19 cases, hospitalizations, and deaths, by vaccination rates and nonpharmaceutical intervention scenarios — United States, April–September 2021," *MMWR. Morbidity and Mortality Weekly Report*, vol. 70, no. 19, pp. 719–724, 2021. [Online]. Available: http://www.cdc.gov/mmwr/volumes/70/wr/mm7019e3.htm?s_cid=mm7019e3_w

[7] Y. Babuji, B. Blaiszik, T. Brettin, K. Chard, R. Chard, A. Clyde, I. Foster, Z. Hong, S. Jha, Z. Li, X. Liu, A. Ramanathan, Y. Ren, N. Saint, M. Schwarting, R. Stevens, H. van Dam, and R. Wagner, "Targeting SARS-CoV-2 with AI- and HPC-enabled lead generation: A first data release," *arXiv:2006.02431 [cs, q-bio, stat]*, 5 2020, arXiv: 2006.02431. [Online]. Available: http://arxiv.org/abs/2006.02431

[8] M. Binois, N. Collier, and J. Ozik, "A portfolio approach to massively parallel Bayesian optimization," *arXiv:2110.09334 [math, stat]*, 10 2021, arXiv: 2110.09334. [Online]. Available: http://arxiv.org/abs/2110.09334

[9] M. de Bayser, L. G. Azevedo, and R. Cerqueira, "ResearchOps: The case for DevOps in scientific applications," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2015, pp. 1398–1404.

[10] M. Feller, I. T. Foster, and S. Martin, "GT4 GRAM: A functionality and performance study," 2007.

[11] B. Enders, D. Bard, C. Snavely, L. Gerhardt, J. Lee, B. Totzke, K. Antypas, S. Byna, R. Cheema, S. Cholia, M. Day, A. Gaur, A. Greiner, T. Groves, M. Kiran, Q. Koziol, K. Rowland, C. Samuel, A. Selvarajan, A. Sim, D. Skinner, R. Thomas, and G. Torok, "Cross-facility science with the Superfacility Project at LBNL," in *2020 IEEE/ACM 2nd Annual Workshop on Extreme-scale Experiment-in-the-Loop Computing (XLOOP)*, 2020, pp. 1–7.

[12] J. Stubbs, R. Cardone, M. Packard, A. Jamthe, S. Padhy, S. Terry, J. Looney, J. Meiring, S. Black, M. Dahan, S. Cleveland, and G. Jacobs, "Tapis: An API platform for reproducible, distributed computational research," in *Advances in Information and Communication*, K. Arai, Ed., 2021, pp. 878–900.

[13] Science Gateways Community Institute, "Creating science gateways success stories," https://sciencegateways.org/app/site/media/files/SGCI_2018_science_highlights_booklet_spreads_FINAL.pdf, 2018.

[14] S. Marru, L. Gunathilake, C. Herath, P. Tangchaisin, M. Pierce, C. Mattmann, R. Singh, T. Gunarathne, E. Chinthaka, R. Gardler, A. Slominski, A. Douma, S. Perera, and S. Weerawarana, "Apache Airavata: A framework for distributed applications and computational workflows," in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, ser. GCE '11.   New York, NY, USA: Association for Computing Machinery, 2011, p. 21–28. [Online]. Available: https://doi.org/10.1145/2110486.2110490

[15] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, 9 2014, [Online; accessed 2021-09-28]. [Online]. Available: http://ieeexplore.ieee.org/document/7036262/

[16] E. Deelman, K. Vahi, M. Rynge, R. Mayani, R. F. da Silva, G. Papadimitriou, and M. Livny, "The evolution of the Pegasus workflow management software," *Computing in Science & Engineering*, vol. 21, no. 4, 2019.

[17] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in Python," ser. HPDC '19.   Phoenix, AZ, USA: Association for Computing Machinery, 6 2019, p. 25–36, [Online; accessed 2020-07-26]. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[18] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *IEEE Congress on Services*, 2007, pp. 199–206.

[19] M. A. Salim, T. D. Uram, J. T. Childers, P. Balaprakash, V. Vishwanath, and M. E. Papka, "Balsam: Automated scheduling and execution of dynamic, data-intensive HPC workflows," *CoRR*, vol. abs/1909.08704, 2019. [Online]. Available: http://arxiv.org/abs/1909.08704

[20] L. Ward, G. Sivaraman, J. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, R. Thakur, and I. Foster, "Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*.   Los Alamitos, CA, USA: IEEE Computer Society, nov 2021, pp. 9–20. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MLHPC54614.2021.00007

[21] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A federated function serving fabric for science."   Stockholm Sweden: ACM, 6 2020, pp. 65–76, [Online; accessed 2021-09-28]. [Online]. Available: https://dl.acm.org/doi/10.1145/3369583.3392683

[22] J. Ozik, N. T. Collier, J. M. Wozniak, and C. Spagnuolo, "From desktop to large-scale model exploration with Swift/T," 12 2016, pp. 206–220, iSSN: 1558-4305.

[23] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing."   Delft: IEEE, 5 2013, pp. 95–102, [Online; accessed 2021-09-28]. [Online]. Available: http://ieeexplore.ieee.org/document/6546066/

[24] G. Pauloski, "ProxyStore: A data fabric for Parsl and FuncX," 2022.

[25] S. Surjanovic and D. Bingham, "Ackley function," http://www.sfu.ca/~ssurjano/ackley.html, 2023, accessed 13th February 2023.

[26] L. Ward, "streaming.py," https://github.com/exalearn/colmena/blob/bd334e0a582fb79d97652d67d05666f13d178f83/demo_apps/optimizer-examples/streaming.py#L1, 2023, accessed 13th February 2023.

[27] A. Al-Saadi, D. H. Ahn, Y. Babuji, K. Chard, J. Corbett, M. Hategan, S. Herbein, S. Jha, D. Laney, A. Merzky, T. Munson, M. Salim, M. Titov, M. Turilli, T. D. Uram, and J. M. Wozniak, "ExaWorks: Workflows for exascale."   St. Louis, MO, USA: IEEE, 11 2021, pp. 50–57, [Online; accessed 2022-07-27]. [Online]. Available: https://ieeexplore.ieee.org/document/9652623/