Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

DISTRIBUTED MODEL EXPLORATION WITH EMEWS

Nicholson Collier[1,2], Justin M. Wozniak[2,3], Arindam Fadikar[1,2], Abby Stevens[1,2], and Jonathan Ozik[1,2]

[1]Decision and Infrastructure Sciences Division, Argonne National Laboratory, Lemont, IL, USA
[2]Consortium for Advanced Science and Engineering, University of Chicago, Chicago, IL, USA
[3]Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA

ABSTRACT

As high-performance computing resources have become increasingly available, new modes of applying and experimenting with simulation and other computational tools have become possible. This tutorial presents recent advancements to the Extreme-scale Model Exploration with Swift (EMEWS) framework. EMEWS is a high-performance computing (HPC) model exploration (ME) framework, developed for large-scale analyses (e.g., calibration, optimization) of computational models. We focus on three new use-inspired EMEWS capabilities, improved accessibility through binary installation, a new decoupled architecture (EMEWS DB) and task API for distributing workflows on heterogeneous compute resources, and improved EMEWS project creation capabilities. We present a complete worked example where EMEWS DB is used to connect a Python Bayesian optimization algorithm to worker pools running both locally and on remote compute resources. The example, including an R version, and additional details on EMEWS are made available on a public website.

1 INTRODUCTION

The Extreme-scale Model Exploration with Swift (EMEWS) framework is a high-performance computing (HPC) model exploration (ME) framework, developed for large-scale analyses of computational models that require the use of approximate, heuristic ME methods, such as calibration or Bayesian optimization, involving large ensembles. EMEWS was introduced to Winter Simulation Conference attendees through a previous advanced tutorial (Ozik et al. 2016). Since that time, the importance and complexity of computational studies for advancing scientific insight and supporting decision making has only increased. Our research group has been applying and evolving EMEWS to adapt to changing requirements related to ease of deployment, robustness, scalability, and heterogeneous computing resources. EMEWS has been used across many computational methods and applications, including agent-based models (ABMs) of infectious diseases (Ozik et al. 2018; Tatara et al. 2019), ABMs of information diffusion (Kaligotla et al. 2020; Lindau et al. 2021), ABMs of molecular systems (Ozik et al. 2019), deep learning for cancer (Wozniak et al. 2018), and microsimulations in cancer (Rutter et al. 2019; Nascimento de Lima et al. 2023).

EMEWS has also been used to support the COVID-19 response in Chicago and Illinois (Ozik et al. 2021). Informed by this experience in time-critical computing, a set of requirements for an open science platform for robust epidemic analysis emerged (Collier et al. 2023). These needs, which include support for very large number of tasks, analyses that extend beyond resource wall times, analysis persistence of data and metadata across computational campaigns, and coordinating tasks across distributed HPC resources, with robust, secure and automated access to each of the resources, guided the development of the new capabilities in EMEWS. In the same pandemic response context, EMEWS has enabled the development of new, fast time-to-solution, large-scale ME algorithms, that can exploit the concurrency provided by HPC systems. These include trajectory-oriented (Fadikar et al. 2023; Fadikar et al. 2024) and large-batch Bayesian optimization (Binois et al. 2021) approaches.

Through this tutorial paper we present the latest advances to the EMEWS framework, with a focus on demonstrating 1) a new software installation approach for ease of deployment, 2) a new decoupled architecture and task API for distributed workflows, and 3) new EMEWS project creation capabilities. These are use-inspired advancements to the EMEWS framework that together enable new and powerful modes of using computational studies in support of advancing science and improving support for evidence based decision making. These and other capabilities can be further explored on the EMEWS Project (2024a) site.

The remainder of this paper is organized as follows. In Section 2, we describe related HPC workflow systems, including how EMEWS expands on existing capabilities. In Section 3, we present the expanded EMEWS capabilities: binary installations, EMEWS DB, and the EMEWS project creator. In Section 4, we provide a complete worked example showing EMEWS DB in both a local and remote configuration, running Python and R ME algorithms. We summarize our contributions and discuss future directions in Section 5.

2 RELATED WORK

Workflow management systems that coordinate tasks on HPC or cloud resources have gained in popularity as many-task computing has played an increasingly central role across scientific domains, with many open source systems being under active development (Workflows Community Initiative 2024). These systems can be thought of as divided between declarative and programmatically defined workflow systems. Merlin (Peterson et al. 2022) is in the first category of systems where a specification, in the form of a directed acyclic graph (DAG), defines all task dependencies and the system is charged with coordinating the necessary steps to complete the set of tasks. Systems like Swift/T (Wozniak et al. 2013) and Parsl (Babuji et al. 2019), on the other hand, use code and code analysis to determine inter-task dependencies. Both approaches have their advantages, including simplifying the workflow description interface for the declarative systems, and enabling more fine-grained control of workflows with the programmatic approach. The EMEWS framework has been built on the Swift/T workflow system (hereafter Swift), which provides flexibility in incorporating multiple programming languages within workflows (Wozniak et al. 2015), and through message passing, compiler techniques (Armstrong et al. 2014), and work stealing capabilities (Lusk et al. 2010), enables EMEWS workflows to efficiently scale to the largest computing resources (Ozik et al. 2021).

Building on the proliferation of workflow systems, there has been an emergence of approaches to allow increasingly complex algorithms to control the overall workflow logic. Two such examples are the Python libraries Colmena (Ward et al. 2021) and libEnsemble (Hudson et al. 2022), where the logic of producing additional tasks based on the results of earlier tasks are encapsulated in Python code called "thinkers" in the former and "generators" in the latter. This *inversion of control* (Ozik et al. 2015) pattern delegates the workflow progression logic to something outside of the workflow system itself and allows for arbitrarily complicated logic that can be defined using, in this case Python-based, scripts and 3rd party libraries for, e.g., statistical or machine learning based decisions on task generation. EMEWS, through the use of general task queues, provides the ability to define such control algorithms, also referred to as *model exploration* (ME) algorithms, in multiple languages, with a primary focus on Python and R. This extends the utility of EMEWS beyond the Python ecosystem (Wozniak et al. 2018; Rutter et al. 2019; Ozik et al. 2021).

Another trend for workflow systems is distributing them across multiple resources. This can provide bursting capabilities when demand for specific tasks within a workflow spike and also can align tasks with their optimal resource configurations. In this space, Globus Compute (formerly funcX) (Chard et al. 2020) has been used by Colmena (Ward et al. 2023) and libEnsemble (Hudson et al. 2024) to securely enable the execution of tasks on a mix of distributed resources, e.g., sending simulation tasks to CPU resources and AI/machine learning tasks to GPU resources. Balsam (Salim et al. 2019) is a Python-based workflow manager that provides centralized access to a number of existing HPC resources, including those at the Argonne and Oak Ridge Leaderships Computing Facilities (ALCF/OLCF) and the National Energy Research Scientific Computing Center (NERSC). Currently, access to the Balsam service is not public but

is administered by the ALCF. With its latest capability enhancements, EMEWS, through the EMEWS DB queues discussed below, leverages Globus Compute to coordinate tasks across federated HPC resources, with robust, secure, and automated access to each of the resources. What is unique about EMEWS is that rather than employ a global task queue, each resource is provided with a local queue, which is located within the security cordon of the resource. This ensures alignment with the diverse authentication requirements of each HPC facility. Furthermore, these capabilities are available to anyone with a free Globus account.

3 EXPANDED EMEWS CAPABILITIES

This section presents new capabilities in the EMEWS framework, including easy-to-install EMEWS binaries (§3.1), the new EMEWS DB decoupled architecture and associated task API for distributing workflows on heterogeneous compute resources (§3.2), and the improved EMEWS project creator (§3.3). Further details on these are found on the EMEWS Project (2024a) site.

3.1 The EMEWS Anaconda Installer

Modern high-level scientific computing environments are used by researchers focused primarily on scientific results, and less focused on computing internals. These can include web or shell-based environments, with scientific tools and libraries that are easy to find, install, and incorporate into workflows. EMEWS is designed as a workflow system that provides access to such scientific resources, but is also intended to be scalable to exotic computing environments in which computing internals are exposed to the user.

To address this deployment challenge, we developed the EMEWS Anaconda Installer, which provides pre-built binaries that can be quickly and seamlessly installed across platforms. This is built on Anaconda-based installations of core EMEWS components, including Swift (Wozniak 2024a), R, Python, and PostgreSQL, and provides further access to Anaconda packages for Python, R, etc., for the user.

The binary installation approach has software engineering trade-offs in terms of maintainability, and strengths and weaknesses from a user perspective, when compared against the alternative, a source-based installation process. The main complexity is that the underlying workflow engine is built on a scalable, MPI and C -based system (Lusk et al. 2010). Compiling this system can produce errors that are unfamiliar to users new to this type of programming. Primarily, this approach pushes the ownership of such issues to the EMEWS team, and decreases the likelihood that end users will see them. This incurs costs on the EMEWS team, as we must maintain binary packages for a range of OS and hardware configurations (currently 4), with associated testing and development workloads.

The strength of this approach is that users can generally get up to speed with EMEWS on a local system as quickly as with other Anaconda-based packages. The potential weakness of the approach is that the user is essentially locked into Anaconda as an "OS." Swift will use the MPI and C language runtime in Anaconda, and other such resources, rather than those that might be available on an exotic computing system such as a tuned vendor-provided MPI installation. However, the binary installation allows for quick and painless startup for experimenting with EMEWS and developing workflows that can be deployed on larger systems with more bespoke software stacks, where a source-based installation of EMEWS may already exist, HPC facility support for installation can be requested, or the user has assessed the utility of EMEWS such that investing in learning about source installations makes sense.

In summary, the EMEWS Anaconda Installer offers a first step for scientific users that desire to quickly access EMEWS capabilities on small-scale resources, in an environment familiar to users of high-level programming environments. Once "hooked" on EMEWS workflows, the user can then progress to larger-scale runs on supercomputing systems supported by the traditional source-based EMEWS installers.

3.2 EMEWS DB

EMEWS DB builds on our earlier queue-like workflow interfaces: EMEWS Queues for Python (EQ/Py) and R (EQ/R) (Ozik et al. 2016). There, the main user interface is a Swift script, a high-level program

that starts the ME algorithm which then provides tasks for evaluation to the Swift script via an in memory output queue. The tasks are distributed by the Swift runtime over a potentially large computer system where they are evaluated, and the results are returned to the ME via an in memory input queue. The two components here, the Swift script and the ME algorithm are tightly coupled. They both run as part of the same HPC job and typically with the same node configuration.

EMEWS DB loosens this tight coupling by separating the execution of the Swift script from that of the ME. A PostgreSQL database serves as a mediator between the two. The ME submits tasks to the EMEWS task database (§3.2.1). A Swift worker pool (§3.2.2) retrieves tasks from the database for evaluation, and returns the results to the database where they are retrieved by the ME. The communication of tasks, their status, and evaluation results are all done through the EMEWS task queue API (§3.2.3). Figure 1 illustrates the main components of EMEWS DB.

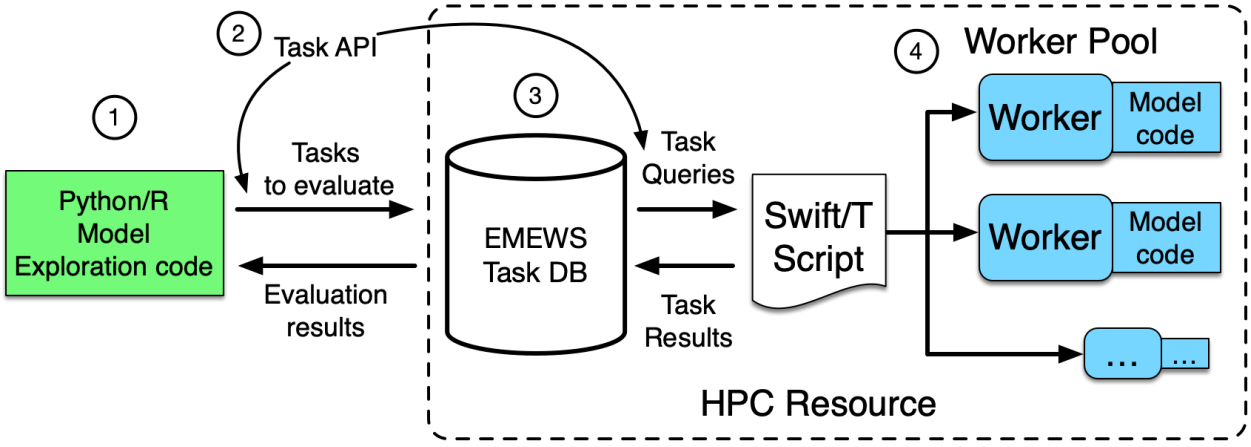

Figure 1: Expanded EMEWS capabilities through EMEWS DB. Python and R-based ME algorithms (1) provide tasks to evaluate via the EMEWS Task API (2). The tasks are added to the EMEWS Task database (DB) (3) and consumed via Swift worker pools (4). Upon task completion, the results are placed back into the EMEWS Task DB, and are made available to the ME algorithms.

While the decoupled nature of EMEWS DB queues adds complexity compared to the existing EQ/R and EQ/Py queues, there are many benefits. These include the ability to run long computational campaigns with very large number of tasks and analyses that extend beyond resource wall times. The more monolithic EQ/R and EQ/Py approaches, while convenient, are limited to the maximum length of time available for individual HPC jobs. In addition, the mediating database better supports the persistence of data and metadata during and across computational campaigns, enabling progress-monitoring tools to access the database, and provides the ability to restart and/or share partial results. Another benefit is that, with increasing levels of heterogeneity and specialization in current and upcoming computing infrastructure, EMEWS DB can enable the execution of different types of work on different hardware resources. For example, ML algorithms are best executed on GPUs or other accelerators, while simulations might better utilize CPU resources. EMEWS DB enables robust, secure, and automated simultaneous access to multiple distributed compute resources, providing the ability to create integrated, multi-resource computational workflows.

3.2.1 EMEWS Task Database

The EMEWS Task Database is a resource local SQL database, consisting of tables and SQL code that implement the task input and output queues, a tasks table (one row for each task), and tables for associated metadata. The tasks table contains the data associated with each task: a unique task identifier; current status (queued, running, completed, 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.

When a task is submitted to the database, an entry for it is created in the tasks table - including the unique id for that task - and the creation time is recorded. Concomitantly, an entry for the task is created in the output queue table. When that task is popped off the output queue table, the task's table entry is updated to tag it as running, and the start time is recorded. When the task has finished and its result is returned to the database, the task's table entry is updated with the result. It is also tagged as completed or canceled, and the stop time is recorded. An entry for the task in the input queue is then created. The task database persists regardless of the state of the worker pool, or ME. For example, an ME can submit tasks, and stop execution. Then when restarted, it can query for the results. Similarly, if a worker pool fails, the task database can be queried to determine the unevaluated tasks, e.g., incomplete tasks due to the worker pool failing, and those can be resubmitted.

Using the Task Queue API (§3.2.3), an ME submits tasks to the database, pushing them on to the output queue, and waits for some number to complete, popping them off the input queue as they become available. Meanwhile, a Swift worker pool also using the Task Queue API, pops tasks off the output queue, evaluates them, and pushes the results back to the input queue. As *resource local* the expectation is that the database will be run on the same resource as the worker pool. This is both to avoid communication overhead between the worker pool, which may be querying the database for 1000's of tasks at a time, and to avoid any security concerns with HPC compute nodes networking with remote databases. EMEWS DB includes an API (the `eqsql.db_tools` module) for starting, stopping and creating the EMEWS task database. This can be used as-is when executed on the resource running the database, or via Globus Compute when the database is on a remote resource. For a more in-depth description of the EMEWS task database, see (Collier et al. 2023).

3.2.2 Swift Heterogeneous Worker Pools

Worker pools are responsible for querying the database via the output queue for submitted tasks, executing them, and then reporting the results of those tasks back to the database where the ME algorithm can retrieve them via the input queue. Our canonical worker pool implementation is a Swift application written using the Swift language and Python, running on a HPC resource (or locally for smaller prototype or debugging runs) as a pilot job. Our worker pool implementation leverages the ability of Swift to distribute work across compute processes and execute that work concurrently, to evaluate as many submitted tasks in parallel as possible. Worker pools can run a variety of task application types using Swift's Python and R interfaces as well as command line executables, including MPI applications, via its `app` function type, and `@par` keyword (Wozniak 2024b).

Worker pools can be matched to those HPC resources configured to most efficiently execute their work type by specifying a work type for each task, such that a worker pool only pops tasks of that type off the output queue. In this way, a worker pool configured to run a multi-process MPI application can retrieve work of that type via an associated work type id versus another work type associated with a single process GPU enabled application, for example. Consequently, the tasks generated by an ME need not be served by a single worker pool, but can leverage increasingly common heterogeneous HPC architectures through differently configured worker pools. EMEWS DB includes a worker pool API (the `eqsql.worker_pool` module) for starting and stopping worker pools both locally and on remote resources.

3.2.3 Task Queue API

The EMEWS DB Task Queue API is implemented in Python in our `eqsql` package, and in R. The R package implementation is essentially a wrapper around the Python code, using Reticulate (Ushey, Allaire, and Tang 2024) together with some bespoke R functions. The core task queue API is defined in a `TaskQueue` Python protocol containing methods for submitting tasks, retrieving results, and for manipulating submitted

tasks (canceling and changing priority). The methods signatures for submitting tasks can be seen in Figure 2 for both individual and lists of tasks.

```
def submit_task(self, exp_id: str, eq_type: int, payload: str, priority: int = 0,
                tag: str = None) -> Tuple[ResultStatus, Union[Future, None]]

def submit_tasks(self, exp_id: str, eq_type: int, payload: List[str],
                 priority: int = 0, tag: str = None) ->
                 Tuple[ResultStatus, List[Future]]
```

Figure 2: EMEWS DB task submission methods.

Tasks are submitted with an experiment id (the `exp_id` argument), identifying the experiment workflow associated with the task, the work type of task (`eq_type`), matching it with a worker pool, a payload string (`payload`), a priority (`priority`), and an optional metadata tag (`tag`). The payload is typically a JSON string containing the input parameters for the task execution. When a task is submitted, an entry is created in the EMEWS database as described in §3.2.1. Task submission returns an instance of an `eqsql.core.Future` or a list of `eqsql.core.Futures`. A `Future` encapsulates the asynchronous execution of a task, and is implemented as a Python class. `Future` class methods allow ME algorithm code to query 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.

Typically an ME algorithm will submit a number of tasks, wait for some number of results, either all of them in the case of an iterative batch algorithm or some subset of the total in the case of an asynchronous algorithm, and then process the results of the task evaluations. The core Task API's `as_completed` method (see Figure 3) allows an ME author to easily implement this pattern.

```
def as_completed(self, futures: List[Future], pop: bool = False,
                 timeout: float = None, n: int = None, batch_size: int = 1,
                 sleep: float = 0) ->
                 Generator[Future, None, None]
```

Figure 3: Task API as_completed method.

`as_completed` takes a Python list of `Futures` and returns a Python generator, allowing client code to iterate over the `Future` tasks as they complete. The `pop` argument, when true, will pop any completed tasks off the list of passed in `Futures`. By default, `as_completed` will loop until all the passed-in tasks have completed. The `n` argument overrides this such that the loop will return that number of completed tasks. The remaining arguments control aspects of the loop iteration, timeout after `timeout` seconds, pause `sleep` amount between querying all the passed in `Futures` for results, and gather `batch_size` number of results before yielding in the generator.

Figure 4 illustrates an example of an asynchronous ME using the Task Queue API. Here, line 6 initializes a local task queue (for more on task queue types, see below). Line 9 creates a list of payloads from 2000 random samples and line 10 submits the payloads for execution. Line 11 creates a `results` dictionary that holds the payload, and result (initially `None`) for each task. This allows the ME to assign the input (a particular task payload) to its evaluation. Then the algorithm iterates for `max_steps` number of iterations, using `as_completed` to retrieve results, popping the completed futures off the list of futures (line 14). The `results` dictionary is updated in line 16 with the new results, and a new set of payloads is created from the updated results in line 18. In lines 19 - 22, the task payloads are submitted, the `results` dictionary is updated, and the new list of `Futures` is added to the existing list passed to `as_completed` in the subsequent step.

```
import numpy as np
import json

from eqsql.task_queues import local_queue

task_queue = local_queue.init_task_queue(...)

rng = np.random.default_rng()
payloads = [json.dumps({'x': for x in rng.random(2000)})]
_, fts = task_queue.submit_tasks(exp_id, task_type, payloads)
results = {ft.eq_task_id: [payloads[i], None] for i, ft in enumerate(fts)}

for step in range(max_steps):
    for ft in task_queue.as_completed(fts, pop=True, n=100):
        result = ft.result()
        results[ft.eq_task_id][1] = result

    payloads = create_payloads(results)
    new_fts = task_queue.submit_tasks(exp_id, task_type, payloads)
    for i, ft in enumerate(new_fts):
        results[ft.eq_task_id] = [payloads[i], None]
    fts += new_fts
```

Figure 4: Sample asynchronous algorithm implementation.

The Task Queue API provides three task queue implementations: one *local* and two *remote*. Here *local* implies that the ME using the task queue code is running on the same resource as the database. The remote task queues use a mediating technology to transfer the payloads and results between the resource where the task queue is executing and the database running on an HPC resource. Both remote task queues ultimately call Python functions on the HPC resource which itself uses a local task queue to submit tasks and retrieve results. The *Globus Compute* task queue uses Globus Compute (Chard et al. 2020), formerly known as funcX, to execute these remote function calls, and the *EMEWS Service* task queue uses JSON messages and a RESTful web service (the EMEWS service) running on the HPC resource that then executes these functions. Regardless of whether the task queue is local or remote, they are all decoupled from worker pool execution such that they can run on a different hardware resource, or with a different configuration.

Local task queues offer many different types of EMEWS DB usage patterns. For example, HPC resources increasingly offer Jupyter notebook servers on which an ME using a local task queue can run, submitting work to a worker pool running on the resource. In addition, HPC job schedulers, such as slurm and PBS, support running a job with different applications and different arguments for each application. In this way, a local ME, using a local task queue, and a worker pool can run as part of the same job, but use different hardware configurations. As for the remote queues, the ME can run on hardware entirely separate from the worker pool(s), in a Jupyter notebook running on a laptop, or on some other hardware resource best suited for the ME, for example, while the worker pools run on more traditional HPC systems.

The above has focused on how an ME uses the Task Queue API to push tasks to the output queue. However, the local queue also contains functionality for popping tasks off the output queue for execution and for pushing the results of executed tasks on to the input queue. This functionality is used by our canonical Swift worker pool implementation to get tasks, execute them concurrently, and push the results back. This capability is implemented only in the local task queue as the EMEWS DB is intended to run locally on the worker pool resource for security and performance as described earlier in §3.2.1. The method signatures for the functionality used by the worker pool can be seen in Figure 5.

`query_more_tasks` is an output queue task query, customized for worker pools. This call allows a worker pool to request up to `batch_size` tasks to consume at a time, while accounting for the number of tasks a worker pool has obtained already but has not completed. So, for example, if a worker pool is

```
def report_task(self, eq_task_id: int, eq_type: int, result: str) -> ResultStatus

def query_more_tasks(self, eq_type: int, eq_task_ids: Iterable[int], batch_size:
```

Figure 5: Worker pool task API.

configured to possess 33 tasks at a time, and if it owns 30 uncompleted tasks when querying the output queue, it will only obtain 3 additional tasks. The `eq_task_ids` argument tracks the tasks currently owned by the worker pool. The task request can be modified 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. The remaining `query_more_tasks` arguments allow a worker pool to query for tasks of a particular type (`eq_type`), tag the task as retrieved by the named worker pool using the `worker_pool` argument, specify the number of seconds to `delay` when polling the EMEWS DB, and a `timeout` value after which the query will timeout. `query_more_tasks` returns a Python Tuple containing a list of the ids of the tasks owned by the worker pool (i.e., those currently running, and any new ones returned by the query), and a list of Python Dictionaries containing the task payloads and query metadata.

The `report_task` method is simple in comparison, and is used to push the result of a task's evaluation to the input queue where it can be retrieved by an ME. The `eq_task_id` is the id of the completed task, `eq_type` the work type , and `result` the result of the task evaluation.

3.3 EMEWS Creator

Implementing an EMEWS workflow can involve writing some amount of boilerplate code, including model and worker pool submission scripts, and ME task submission and retrieval. EMEWS Creator, a Python command line application for creating EMEWS workflow projects, automates the creation of this boiler plate code for input parameter sweep, EQ/Py, EQ/R, and EMEWS DB projects. It can also be used to initialize an EMEWS task database. As a Python package, it can be installed from PyPI using `pip install emewscreator`.

The `emewscreator` command line application takes five commands: 1) `eqpy`, which creates an EQ/Py workflow project; 2) `eqr`, which creates an EQ/R workflow project; 3) `sweep`, which creates a sweep workflow project; 4) `eqsql`, which creates an EMEWS DB workflow project; and 5) `initdb`, which initializes an EMEWS task database. Each of the project commands can be configured using a yaml format file or through command line arguments. To create an EMEWS DB project, for example, configured by a "my_cfg.yaml" file in an "emews" directory, the command line would look like:

```
emewscreator -o emews eqsql -c my_cfg.yaml
```

Example configuration files can be found on the EMEWS Creator repository (EMEWS Project 2024c).

Running `emewscreator` creates a canonical project directory tree containing Python, R, and Swift code, as well as bash scripts for running a model, and submitting Swift workflows to local and HPC resources. For example, creating an EMEWS DB project will produce Python and R prototype ME code, and a Swift worker pool implementation, together with other required files (Swift extensions, and submission scripts). The Python and R prototype code includes boiler plate for starting the EMEWS task database, initializing the task queue, checking for a coherent input and output queue state, and starting a worker pool. Sections marked with *TODO* contain examples of the Task API (§3.2.3) for creating task payloads, submitting those payloads, and retrieving results. Other files also include *TODO* sections, but much of the customization is achieved using yaml format configuration files, prototypes of which are also produced by

EMEWS Creator. A much more thorough discussion of EMEWS Creator and all its options can be found in the EMEWS Tutorial at the EMEWS Project (2024a) site.

4 EMEWS DB WORKED EXAMPLE

In order to showcase the various elements of EMEWS DB, we provide a complete worked example where we use a simple simulation model and an ME algorithm to optimize it. We first present the details of the simulation model and ME algorithm (§4.1) and then demonstrate how to use the capabilities of EMEWS DB to create local and remote Bayesian optimization EMEWS workflows (§4.2).

4.1 Simulation Model and ME Algorithm

For this example, we have adapted the Zombies demonstration model distributed with Repast4Py (Collier and Ozik 2022). The Zombies model involves two agent types, Zombies and Humans, where the Zombies pursue Humans, seeking to infect them. Once a Human agent is infected it is transformed into a Zombie agent after an incubation period lasting a number of time steps. Each time step, each Zombie and Human examines their local Moore neighborhood and moves towards the location with the most Humans or fewest Zombies, respectively. In this adaptation, we have introduced a varying movement step size for each of the agent types. The original model had Zombies move in fixed steps of length 0.25 (in units of the model space) and Humans in steps of length 0.5. The present model encapsulates these two values in two float type parameters `zombie_step_size` and `human_step_size`.

The value to optimize in this example is the remaining number of Humans at a specific simulation time step, where we deem more surviving Humans as a better outcome. Since this is a stochastic model, we run each parameter combination (zombie_step_size, human_step_size) five times, varying the random seed for each, and the remaining number of Humans is averaged to yield the final number of survivors. The adaptive parameter search algorithm we used is Thompson sampling (TS). TS (Thompson 1933; Thompson 1935) is a probabilistic approach used in decision-making under uncertainty, particularly in the context of sequential decision problems like reinforcement learning (Russo et al. 2018), multi-armed bandit problems (Agrawal and Goyal 2012), and Bayesian optimization (Fadikar et al. 2023). In TS, decisions are made by sampling from the posterior distribution of the unknown parameters in the problem space. In the Zombies model, this distribution represents the expected number of surviving Humans based on different step sizes taken by Humans and Zombies. The key idea is to balance exploration (trying new options) and exploitation (leveraging current knowledge) for making optimal decisions by sampling from the posterior distribution of the simulation input at untested settings. Here we integrate the TS algorithm with a Gaussian process (GP) (Williams and Rasmussen 2006) surrogate, providing a non-parametric regression model to anticipate simulation results at unexplored configurations with probabilistic uncertainty, and a method to efficiently sample multiple parameters at a time (TS). To start with, a GP model is trained on an initial batch of simulations and subsequently updated as the simulation dataset expands during optimization. At each iteration, samples are drawn from the fitted GP's predictive surface to identify the next batch of parameter combinations for conducting new simulations. The pseudo-code for the ME can be seen in Algorithm 1.

Figure 6 demonstrates that the TS procedure is able to find an optimal parameter configuration after just two iterations. We first fit a GP using an evenly spaced initial design of size 5. The resulting predictive surface suggests that higher values (i.e., the maximum number of surviving Humans) are more likely at parameters near the origin (bottom left corner). The TS procedure then samples new points close to the origin. We fit another GP and notice that the surface is smoother, especially for smaller Zombie step sizes. We perform one more TS iteration and only find a few additional points, evidence that the first iteration managed to characterize the parameter space quite well already. The results confirm that, for Humans to have the best chance of survival, Zombies should move as slowly as possible while Humans must move faster but not too much faster, staying within their examined local neighborhood and not running into unseen areas where Zombies might exist.

Algorithm 1 Pseudo-code for the TS and GP ME algorithm used to optimize the Zombies model.

```
Create the initial samples
Submit the samples for evaluation by Zombies model
Wait for results (average number of Humans)
Update the GP surrogate with the results
for step in n_steps do
    Re-sample using TS
    Submit the samples for evaluation by Zombies model
    Wait for results (average number of Humans)
    Update the GP surrogate with the results
end for
```

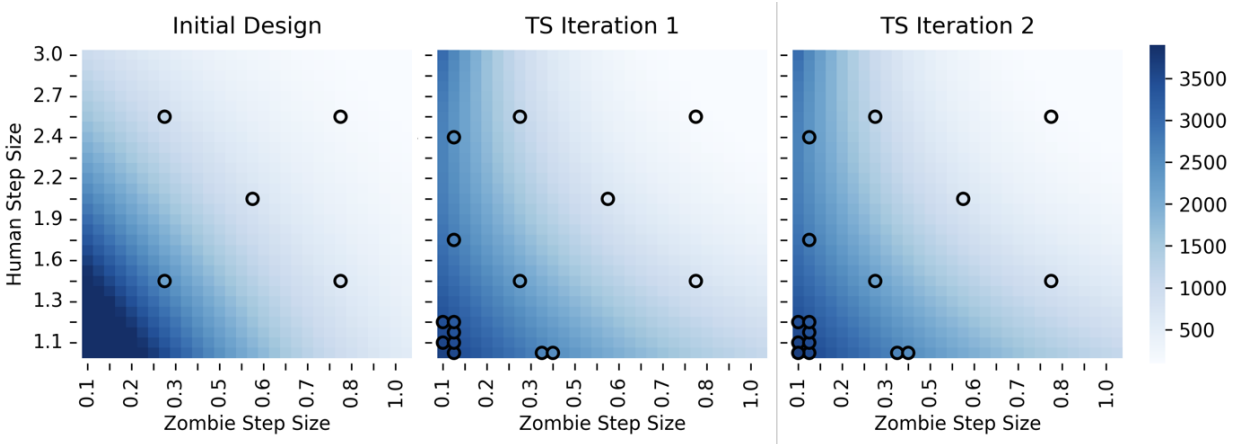

Figure 6: **Bayesian Optimization of the Zombies model.** The left plot shows the initial simulation design (in black circles), with data points sampled evenly across the grid; the middle plot is after the first TS iteration, and the right plot is after the second TS iteration. The heatmap shows the predicted mean model output from a fitted GP surrogate. The optimization algorithm finds the optimal parameter configuration that maximizes surviving Humans.

4.2 EMEWS DB Example Workflows

The pseudo-code from Algorithm 1 is implemented in Python in a Jupyter notebook, which can be found in the tutorial example repository (EMEWS Project 2024b). Leaving aside the specifics of the TS implementation and the plotting code used to produce Figure 6, the following will focus on the EMEWS DB specific parts of the notebook implementation. This first example uses a local task queue (§3.2.3) and subsequently we demonstrate how to extend it to a Globus Compute remote task queue where tasks are run on a remote HPC resource. Figure 7 begins with the initialization of the various EMEWS DB components.

Here `params` is a Python dictionary containing the configuration parameters for the ME. This is typically loaded from a yaml format file created using EMEWS Creator (§3.3). The initialization begins by starting the database (line 4) and initializing a local task queue (line 5) using the EMEWS DB `eqsql` package imported in lines 1 and 2. The `task_queue` will submit tasks and retrieve results for the queues in that database. In line 7 we check if the input and output database queues are empty and, if not, we abort the run, ensuring that only tasks produced in this run are submitted and retrieved. Having confirmed that the input and output queues are in a coherent state, a Swift worker pool is started, again using a Python module from the `eqsql` packages. In this case, a local pool is started, where *local* indicates that the pool runs on the same resource as the ME (i.e., the one running the notebook). Whether local or remote, the pool

```
from eqsql import worker_pool, db_tools
from eqsql.task_queues import local_queue
...
db_tools.start_db(params['db_path'], pg_bin_path=params['pg_bin_path'])
task_queue = local_queue.init_task_queue(params['db_host'], params['db_user'],
                                         port=None, db_name=params['db_name'])
if not task_queue.are_queues_empty():
    print("WARNING: db input / output queues are not empty. Aborting run")
    return

pool_params = worker_pool.cfg_file_to_dict(params['pool_cfg_file'])
pool = worker_pool.start_local_pool(params['worker_pool_id'],
                                    params['pool_launch_script'], exp_id,
                                    pool_params)
```

Figure 7: EMEWS DB initialization.

will query the output queue of the database, popping tasks off the queue, and, with as much concurrency as it is configured for (see §3.2.2), pass each task payload as input to a Zombies model run.

Once the EMEWS DB components have been initialized, the initial sample values can be submitted and their results are used to train the GP and produce more points for evaluation from the TS. In Figure 8, 5 initial demonstration samples, `X`, hard-coded as indices in a 30×30 grid, `Xgrid`, are created (lines 1 and 2), and scaled for Zombies model step sizes from their $0-1$ range to $0.1-1$ for Zombies, and $1-3$ for Humans in line 3. The resulting `Xnative` variable is a two-dimensional NumPy array where each row in the array contains two elements, a Zombie step size, and a Human step size. The task payloads are created from `Xnative` in line 5. Each element in `Xnative` is transformed into a Python List and that list is then formatted as a JSON string. The local task queue then submits `payloads` for evaluation by the worker pool (line 6). The task submission returns a Python Tuple containing the results of the task submission (i.e., success or failure) and a list of `Future` objects representing each of those tasks. Line 7 passes `fts` (the `Futures`), to `as_completed`, which returns each future as its task completes, and sorts the results by task id. Sorting by task id, orders the completed `Futures` in the order they were submitted, that is, in the same order as `Xnative`, aligning the task result order with that of the inputs. Line 8 uses a list comprehension to create a NumPy array of the results, by transforming each result from JSON format to a floating point number. We then take the `log` of each of those results for numerical stability, train the GP on the results, and begin the ME loop.

```
init_ids = [217, 233, 465, 697, 713]
X, Xgrid = initial_samples(init_id=init_ids)
Xnative = to_native(X, lb=np.array([0.1, 1]), ub=np.array([1, 3]))

payloads = list(map(lambda a: json.dumps(list(a)), Xnative))
_, fts = task_queue.submit_tasks(exp_id, task_type, payloads)
sorted_fts = sorted(task_queue.as_completed(fts), key=lambda ft: ft.eq_task_id)
Y = np.array([json.loads(ft.result()[1]) for ft in sorted_fts])
# log response for numerical stability
Y = np.log(Y)
...
# fit GP on logged response
gp = GPy.models.GPRegression(X, Y.reshape(-1,1))
gp.optimize()
```

Figure 8: Initial samples submission and retrieval.

The ME loop looks much the same as the evaluation of the initial samples, and can be seen in Figure 9. The looped code begins in line 5 where the TS produces `n_points` new points for evaluation. These points are then scaled to Zombies model step sizes in line 6. Lines 8 through 13 submit the new step sizes and format the results as in the initial sample evaluation. The GP is then updated with the new results in lines 16 through 21.

```
n_steps = params['n_steps']
n_points = params['n_points']

for i in np.arange(1, n_steps + 1):
    X_new = TS_npoints(model = gp, npoints = n_points, Xgrid = Xgrid)
    X_new_native = to_native(X_new, lb=np.array([0.1, 1]), ub=np.array([1, 3]))

    payloads = list(map(lambda a: json.dumps(list(a)), X_new_native))
    _, fts = task_queue.submit_tasks(exp_id, task_type, payloads)
    sorted_fts = sorted(task_queue.as_completed(fts), key=lambda ft: ft.eq_task_id)
    Y_new = np.array([json.loads(ft.result()[1]) for ft in sorted_fts])
    # log for numerical stability
    Y_new = np.log(Y_new)

    # append to existing results
    X = np.vstack([gp.X, X_new])
    Y = np.vstack([gp.Y, Y_new.reshape(-1,1)])

    # update GP with new data
    gp = GPy.models.GPRegression(X, Y.reshape(-1,1))
    gp.optimize()
```

Figure 9: ME sample and submission loop.

The notebook implementation, described above, shows how to use EMEWS DB with a local task queue, and was coded using a local environment created by the EMEWS Anaconda Installer (§3.1), where the TS and GP requirements were installed using `conda` and `pip`. To run the notebook with a remote task queue and worker pool, very little code needs to be changed. The task queue and worker pool initialization in Figure 7 need to be updated to use their remote equivalents. Figure 10 illustrates the initialization of a Globus Compute task queue, and a remote worker pool on an HPC resource. Line 1 imports the `gc_queue` module that contains the Globus Compute task queue. The `gc_queue` and the remote worker pool initialization both require a Globus Compute `Executor`. Line 3 imports the required package, and lines 5 and 6 create the `Executor`. An `Executor` is best used within a Python context block, and lines 6 through 11 reflect that. The new `gc_queue` is created in line 7, taking the same arguments as the local queue with the addition of the `Executor` (`gcx`). The remote worker pool is started in line 11, taking the same arguments as the local version, with the addition of the HPC scheduler type (`slurm`), and the `Executor` (`gcx`). With these changes in place, the rest of the implementation remains the same. The only additional changes are that the worker pool configuration must be updated for the remote HPC resource and the EMEWS DB needs to be started there.

Here we presented local and remote EMEWS DB workflows implemented using the Python Task API. A complete worked example showing the same examples, but using the R Task API is available on the tutorial example repository (EMEWS Project 2024b).

5 SUMMARY AND FUTURE WORK

The use-inspired development of new EMEWS capabilities was guided by the evolving requirements in computational studies, whether they be related to ease of deployment, robustness, scalability, or the use of

```
from eqsql.task_queues import gc_queue
from eqsql import worker_pool
import globus_compute_sdk

gc_endpoint = 'xxxxxx'
with globus_compute_sdk.Executor(gc_endpoint) as gcx:
    task_queue = gc_queue.init_task_queue(gcx,
                                          params['db_host'],params['db_user'],
                                          port=None, db_name=params['db_name'])
    pool_params = worker_pool.cfg_file_to_dict(params['pool_cfg_file'])
    pool = worker_pool.start_scheduled_pool(params['worker_pool_id'],
                                            params['pool_launch_script'], exp_id,
                                            pool_params, 'slurm', gcx)
    ...
```

Figure 10: ME with remote worker pool.

heterogeneous computing resources. In this tutorial we have described these expanded capabilities: improved accessibility through a new Anaconda binary installer, the new decoupled EMEWS DB architecture and task API for distributing workflows on heterogeneous compute resources, and improved EMEWS project creation capabilities. We have also demonstrated the use of EMEWS DB in a simple optimization example that exercises various parts of its API. Our hope is that these capabilities are able to further advance computational science and improve how it can support evidence based decision making.

Future work will focus on additional performance optimizations, particularly for the `as_completed` method. Currently, completed tasks are retrieved one at a time, each requiring a database access. This overhead is minimal with lower tasks counts, but can become significant at larger (100K+) counts. By batching the completed task retrieval into much fewer database calls, much, if not all, of this overhead could be eliminated. A similar optimization for batching task writes to the output queue is also planned.

6 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 2200234, the National Institutes of Health under grants R01AI158666, U01CA253913, R01DA057350, and R01DA055502, the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357, and the DOE Office of Science through the Bio-preparedness Research Virtual Environment (BRaVE) initiative. This research was completed with resources provided by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility, the National Energy Research Scientific Computing Center (NERSC), the Laboratory Computing Resource Center at Argonne National Laboratory, and the Research Computing Center at the University of Chicago.

REFERENCES

Agrawal, S. and N. Goyal. 2012. "Analysis of Thompson Sampling for the Multi-armed Bandit Problem". In *Proceedings of the 25th Annual Conference on Learning Theory*, edited by S. Mannor, N. Srebro, and R. C. Williamson, Volume 23 of *Proceedings of Machine Learning Research*, 39.1–39.26. Edinburgh, Scotland: PMLR https://doi.org/10.5220/0005184400550065.

Armstrong, T. G., J. M. Wozniak, M. Wilde, and I. T. Foster. 2014. "Compiler Techniques for Massively Scalable Implicit Task Parallelism". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, 299–310. Piscataway, NJ, USA: IEEE Press https://doi.org/10.1109/SC.2014.30.

Babuji, Y., A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, , , , , *et al*. 2019. "Parsl: Pervasive Parallel Programming in Python". In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, 25–36. Phoenix, AZ, USA: Association for Computing Machinery https://doi.org/10.1145/3307681.3325400.

Binois, M., N. Collier, and J. Ozik. 2021. "A portfolio approach to massively parallel Bayesian optimization". Technical report.

Chard, R., Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, *et al*. 2020. "funcX: A Federated Function Serving Fabric for Science". 65–76. Stockholm Sweden: ACM https://doi.org/10.1145/3369583.3392683. [Online; accessed 2021-09-28].

Collier, N. and J. Ozik. 2022. "Distributed Agent-Based Simulation with Repast4Py". In *2022 Winter Simulation Conference (WSC)*, 192–206 https://doi.org/10.1109/WSC57314.2022.10015389.

Collier, N., J. M. Wozniak, A. Stevens, Y. Babuji, M. Binois, A. Fadikar, , *et al*. 2023. "Developing Distributed High-performance Computing Capabilities of an Open Science Platform for Robust Epidemic Analysis". In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 868–877 https://doi.org/10.1109/IPDPSW59300.2023.00143.

EMEWS Project 2024a. "EMEWS". https://emews.github.io. accessed 03rd May 2024.

EMEWS Project 2024b. "EMEWS DB Example Tutorial Github site". https://github.com/emews/emews_tutorial_BO. accessed 03rd May 2024.

EMEWS Project 2024c. "EMEWS Project Creator Github site". https://github.com/emews/emews-project-creator. accessed 03rd May 2024.

Fadikar, A., N. Collier, A. Stevens, J. Ozik, M. Binois and K. B. Toh. 2023. "Trajectory-Oriented Optimization of Stochastic Epidemiological Models". In *2023 Winter Simulation Conference (WSC)*, 1244–1255. San Antonio, TX, USA: IEEE https://doi.org/10.1109/WSC60868.2023.10408258.

Fadikar, A., A. Stevens, N. Collier, K. B. Toh, O. Morozova, A. Hotton, , *et al*. 2024. "Towards Improved Uncertainty Quantification of Stochastic Epidemic Models Using Sequential Monte Carlo". Technical report. arXiv:2402.15619 [stat].

Hudson, S., J. Larson, J.-L. Navarro, and S. M. Wild. 2022. "libEnsemble: A Library to Coordinate the Concurrent Evaluation of Dynamic Ensembles of Calculations". *IEEE Transactions on Parallel and Distributed Systems* 33(4):977–988 https://doi.org/10.1109/TPDS.2021.3082815.

Hudson, S., J. Larson, J.-L. Navarro, and S. M. Wild. 2024. "Portable, heterogeneous ensemble workflows at scale using libEnsemble". Technical report https://doi.org/10.48550/arXiv.2403.03709. arXiv:2403.03709 [cs].

Kaligotla, C., J. Ozik, N. Collier, C. M. Macal, K. Boyd, J. Makelarski, *et al*. 2020. "Model Exploration of an Information-Based Healthcare Intervention Using Parallelization and Active Learning". *Journal of Artificial Societies and Social Simulation* 23(4):1 https://doi.org/10.18564/jasss.4379.

Lindau, S. T., J. A. Makelarski, C. Kaligotla, E. M. Abramsohn, D. G. Beiser, C. Chou, , , , *et al*. 2021. "Building and experimenting with an agent-based model to study the population-level impact of CommunityRx, a clinic-based community resource referral intervention". *PLOS Computational Biology* 17(10):e1009471 https://doi.org/10.1371/journal.pcbi.1009471.

Lusk, E. L., S. C. Pieper, and R. M. Butler. 2010. "More scalability, less pain : A simple programming model and its implementation for extreme computing.". *SciDAC Rev.* 17(2010):30–37. Institution: Argonne National Lab. (ANL), Argonne, IL (United States) Number: ANL/MCS/JA-65869.

Nascimento de Lima, P., R. van den Puttelaar, A. I. Hahn, M. Harlass, N. Collier, J. Ozik, , *et al*. 2023. "Projected long-term effects of colorectal cancer screening disruptions following the COVID-19 pandemic". *eLife* 12:e85264 https://doi.org/10.7554/eLife.85264. Publisher: eLife Sciences Publications, Ltd.

Ozik, J., N. Collier, R. Heiland, G. An and P. Macklin. 2019. "Learning-accelerated discovery of immune-tumour interactions". *Molecular Systems Design & Engineering* 4(4):747–760 https://doi.org/10.1039/C9ME00036D.

Ozik, J., N. T. Collier, and J. M. Wozniak. 2015. "Many Resident Task Computing in Support of Dynamic Ensemble Computations". In *8th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers Proceedings*. http://datasys.cs.iit.edu/events/MTAGS15/program.html.

Ozik, J., N. T. Collier, J. M. Wozniak, C. M. Macal and G. An. 2018. "Extreme-Scale Dynamic Exploration of a Distributed Agent-Based Model With the EMEWS Framework". *IEEE Transactions on Computational Social Systems* 5(3):884–895 https://doi.org/10.1109/TCSS.2018.2859189.

Ozik, J., N. T. Collier, J. M. Wozniak, and C. Spagnuolo. 2016. "From desktop to Large-Scale Model Exploration with Swift/T". In *2016 Winter Simulation Conference (WSC)*, 206–220 https://doi.org/10.1109/WSC.2016.7822090.

Ozik, J., J. M. Wozniak, N. Collier, C. M. Macal and M. Binois. 2021. "A population data-driven workflow for COVID-19 modeling and learning". *The International Journal of High Performance Computing Applications* 35(5):483–499 https://doi.org/10.1177/10943420211035164.

Peterson, J. L., B. Bay, J. Koning, P. Robinson, J. Semler, J. White, , , , , , , , , , , , *et al*. 2022. "Enabling machine learning-ready HPC ensembles with Merlin". S0167739X22000322 https://doi.org/10.1016/j.future.2022.01.024.

Russo, D. J., B. Van Roy, A. Kazerouni, I. Osband, Z. Wen *et al*. 2018. "A tutorial on thompson sampling". *Foundations and Trends® in Machine Learning* 11(1):1–96 https://doi.org/10.1561/9781680834710.

Rutter, C. M., J. Ozik, M. DeYoreo, and N. Collier. 2019. "Microsimulation model calibration using incremental mixture approximate Bayesian computation". *The Annals of Applied Statistics* 13(4):2189–2212 https://doi.org/10.1214/19-AOAS1279.

Salim, M. A., T. D. Uram, J. T. Childers, P. Balaprakash, V. Vishwanath and M. E. Papka. 2019. "Balsam: Automated Scheduling and Execution of Dynamic, Data-Intensive HPC Workflows". *CoRR* abs/1909.08704. http://arxiv.org/abs/1909.08704.

Tatara, E., N. T. Collier, J. Ozik, A. Gutfraind, S. J. Cotler, H. Dahari, *et al*. 2019. "Multi-Objective Model Exploration of Hepatitis C Elimination in an Agent-Based Model of People who Inject Drugs". In *2019 Winter Simulation Conference (WSC)*, 1008–1019 https://doi.org/10.1109/WSC40007.2019.9004747. ISSN: 0891-7736.

Thompson, W. R. 1933. "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples". *Biometrika* 25(3/4):285 https://doi.org/10.2307/2332286.

Thompson, W. R. 1935. "On the Theory of Apportionment". *American Journal of Mathematics* 57(2):450 https://doi.org/10.2307/2371219.

Ushey, K., J. Allaire, and Y. Tang. 2024. *reticulate: Interface to 'Python'*. R package version 1.36.1, https://rstudio.github.io/reticulate/.

Ward, L., J. G. Pauloski, V. Hayot-Sasson, R. Chard, Y. Babuji, G. Sivaraman, , , *et al*. 2023. "Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources". 32–41: IEEE Computer Society https://doi.org/10.1109/IPDPSW59300.2023.00018.

Ward, L., G. Sivaraman, J. G. Pauloski, Y. Babuji, R. Chard, N. Dandu, , , , , *et al*. 2021. "Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing". In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, 9–20: IEEE https://doi.org/10.1109/MLHPC54614.2021.00007.

Williams, C. K. and C. E. Rasmussen. 2006. *Gaussian processes for machine learning*, Volume 2. MIT press Cambridge, MA https://doi.org/10.7551/mitpress/3206.003.0005.

Workflows Community Initiative 2024. "Workflows Systems Under Active Development". https://workflows.community/systems.

Wozniak, Justin M. 2024a. "Swift/T Anaconda Installation". https://anaconda.org/swift-t. accessed 10th May 2024.

Wozniak, Justin M. 2024b. "Swift/T User Guide". https://swift-lang.github.io/swift-t/guide.html. accessed 10th May 2024.

Wozniak, J. M., T. G. Armstrong, K. Maheshwari, D. S. Katz, M. Wilde and I. T. Foster. 2015. "Interlanguage Parallel Scripting for Distributed-memory Scientific Computing". In *Proceedings of the 10th Workshop on Workflows in Support of Large-Scale Science*, WORKS '15, 6:1–6:11. New York, NY, USA: ACM https://doi.org/10.1145/2822332.2822338.

Wozniak, J. M., T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk and I. T. Foster. 2013. "Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing". In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 95–102: IEEE https://doi.org/10.1109/CCGrid.2013.99.

Wozniak, J. M., R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, , , , , , *et al*. 2018. "CANDLE/Supervisor: a workflow framework for machine learning applied to cancer research". *BMC Bioinformatics* 19(18):491 https://doi.org/10.1186/s12859-018-2508-4.

AUTHOR BIOGRAPHIES

NICHOLSON COLLIER is a Senior Software Engineer at Argonne National Laboratory, and Staff Software Engineer in the Consortium for Advanced Science and Engineering at the University of Chicago. As the lead developer for the Repast project for agent-based modeling toolkits and co-developer of the Extreme-scale Model Exploration with Swift (EMEWS) framework, he develops, architects, and implements large-scale agent-based models and frameworks, and large-scale model exploration workflows across various domains. His e-mail address is ncollier@anl.gov.

JUSTIN M. WOZNIAK is a computer scientist at Argonne National Laboratory, and scientist-at-large in the Consortium for Advanced Science and Engineering at the University of Chicago. His research interests include programming systems for high performance computing and scientific uses of machine learning. He is the lead developer of the Swift/T workflow language. His email address is woz@anl.gov.

ARINDAM FADIKAR is an assistant computational statistician in the Decision and Infrastructure Sciences division at Argonne National Laboratory. His primary research interest is in the area of design and analysis, calibration, uncertainty quantification of computer models under input-dependent noise with application in epidemiology, material science, urban traffic network systems, and cosmology. His email address is afadikar@anl.gov.

ABBY STEVENS is a computational data scientist at Argonne National Laboratory. Her research interests include interpretable machine learning, surrogate modeling, and data assimilation with applications in epidemiology and climate science. Her email address is stevensa@anl.gov.

JONATHAN OZIK is a Principal Computational Scientist at Argonne National Laboratory, Senior Scientist with Department of Public Health Sciences affiliation in the Consortium for Advanced Science and Engineering (CASE) at the University of Chicago, and Senior Institute Fellow in the Northwestern Argonne Institute of Science and Engineering (NAISE) at Northwestern University. He is the lead of the Repast project for agent-based modeling toolkits and the Extreme-scale Model Exploration with Swift (EMEWS) framework for large-scale model exploration on high-performance computing resources. His e-mail address is jozik@anl.gov.