FaaSr: Cross-Platform Function-as-a-Service Serverless Scientific Workflows in R

Sungjae Park
Department of Electrical and Computer Engineering
University of Florida
Gainesville, Florida, USA
spark7@ufl.edu

R. Quinn Thomas
Department of Forest Resources and Environmental Conservation
Virginia Tech
Blacksburg, Virginia, USA
rqthomas@vt.edu

Cayelan C. Carey
Department of Biological Sciences
Virginia Tech
Blacksburg, Virginia, USA
cayelan@vt.edu

Austin D. Delany
Department of Biological Sciences
Virginia Tech
Blacksburg, Virginia, USA
addelany@vt.edu

Yun-Jung Ku
Department of Electrical and Computer Engineering
University of Florida
Gainesville, Florida, USA
y.ku@ufl.edu

Mary E. Lofton
Department of Biological Sciences
Virginia Tech
Blacksburg, Virginia, USA
melofton@vt.edu

Renato J. Figueiredo†
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon, USA
figueren@oregonstate.edu

2024 IEEE 20th International Conference on e-Science (e-Science) | 979-8-3503-6561-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/e-Science62913.2024.10678660

***Abstract*—Modern Function-as-a-Service (FaaS) cloud platforms offer great potential for supporting event-driven scientific workflows. Nonetheless, there remain barriers to adoption by the scientific community in domains such as environmental sciences, where R is the focal language used for the development of applications and where users are typically not well-versed with FaaS APIs. This paper describes the design and implementation of FaaSr, a novel middleware system that supports event-driven scientific workflows in R. A key novelty in FaaSr is the ability to deploy workflows across FaaS providers without the need for any managed servers for coordination. With FaaSr: 1) functions are written in R; 2) the runtime environments for their execution are customizable containers; 3) functions access data in cloud storage (S3) with a familiar file-based abstraction supporting both full file put/get primitives and subsetting using the Parquet format; and 4) function invocation and workflow coordination only requires S3 cloud object storage, without relying on any dedicated, active workflow engine server or cloud-specific queues/databases. The paper reports on the functionality and performance of FaaSr for micro-benchmarks and two case studies: event-driven forecast and batch job workflows. These demonstrate the ability to deploy workflows across multiple platforms (GitHub Actions, Amazon Web Services Lambda, and the open-source OpenWhisk), without the need for dedicated coordination servers, across both cloud and edge resources. FaaSr is open-source and available as a CRAN package.**

***Index Terms*—cloud, cyberinfrastructure, Function-as-a-Service, serverless, workflow**

This work is supported by the US National Science Foundation (NSF) as part of awards OAC-2311123, OAC-2311124, EF-2318861 EF-2318862, DBI-1933102, and DBI-1933016. This work used CloudLab resources and Jetstream2 at Indiana University through allocation DEB170011 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by NSF grants 2138259, 2138286, 2138307, 2137603, and 2138296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.
†Work primarily performed while the author was at the Department of Electrical and Computer Engineering at the University of Florida.

I. Introduction

As scientific discoveries are increasingly reliant on computational models and driven by data observations, there is growing demand for cyberinfrastructure that is accessible to developers and end users across various scientific domains. Modern cloud platforms for computing and storage hold great potential to meet this demand: in particular, Function-as-a-Service (FaaS) serverless computing offers scalable, on-demand distributed computing, while Simple Storage Service (S3) offers scalable, resilient distributed storage. The FaaS serverless approach is especially appealing for event-driven execution of scientific workflows [1]–[7], as the user does not need to be burdened with the management of any cloud servers (hence the term serverless) and computation can be dynamically triggered as soon as data become available.

However, FaaS platforms have been originally designed to target Web-based applications, and there is still a major adoption gap in the scientific community. In particular, this gap is notable for domains such as the environmental sciences, where R is the focal language used for the development of applications and where users are typically not well-versed with FaaS APIs. Furthermore, while several FaaS platforms exist, they expose different, non-compatible interfaces. As a result, if users develop workflows for a particular platform, they can become locked-in.

To address this gap, this paper describes the design and implementation of FaaSr, a novel open-source middleware that supports event-driven scientific workflows with several key features. With FaaSr: 1) functions are written in the R language; 2) functions are composed into Directed Acyclic Graph (DAG) workflows; 3) the runtime environments are containers that can be customized per function and reproducibly deployed

across multiple FaaS providers; 4) functions access persistent data on S3 cloud storage with simple file I/O interfaces, and 5) all workflow coordination is accomplished using passive S3 cloud storage, requiring no active coordination servers.

FaaSr allows users to write their function code once, using both a familiar language and a familiar file-based abstraction, without any knowledge of FaaS-specific APIs. *As a result, it lowers the barrier to entry for scientific workflows to use event-driven serverless computing*. Furthermore, FaaSr does not depend on any managed servers or provider-specific synchronization and messaging queues. *As a result, it providers users with deployment choice rather than vendor lock-in*. In short, all that a user needs to deploy FaaSr workflows are: 1) accounts and credentials with one or more FaaS/S3 providers, 2) R functions available in a Git repository, and 3) workflow configurations in JSON format.

FaaSr accomplishes its goals by: 1) abstracting users from FaaS-specific APIs by exposing simple interfaces and encapsulating FaaS-specific action invocation, payload handling, and action triggers in "stubs" transparent to users; 2) integrating with a well-established community container (Rocker) in the absence of cloud-native R runtimes; and 3) coordinating DAG execution among concurrent tasks using file-based locks and concurrency control atop passive S3 cloud storage without any providers-specific infrastructure that could be subject to vendor lock-in. This paper makes the following contributions:

- To the best of our knowledge, FaaSr is the first system to enable serverless DAG workflows across different FaaS platforms where: 1) no dedicated workflow server/VM is required, and 2) each individual function can run on a different FaaS platform. Together, *these two capabilities are key to lowering barriers to adoption because users: a) do not need to deploy or manage any workflow engines or servers, and b) are not subject to vendor lock-in*. These barriers are challenging for individual users and small research groups in "long end tail of science" and in disciplines such as environmental sciences, where the overheads associated with deploying/managing servers/VMs and developing for FaaS-specific APIs is a key deterrent.
- We describe a novel design for cross-platform workflow coordination among actions in a DAG that allows actions receiving multiple triggers from N predecessors in the graph to self-select whether to abort ($N-1$ actions) or continue (single action) using passive S3 object storage for coordination, without relying on cloud-specific messaging queues. This reduces dependencies on specific cloud providers and preventing vendor lock-in.
- We describe an implementation of FaaSr that has been released as open-source software packaged and available for R users through the widely-used CRAN repository [8]. While currently implemented in R, the FaaSr architecture is generalizable to other languages.
- We evaluate its performance with micro-benchmarks and application workflows using three different serverless platforms that cater to a broad set of use cases: GitHub Actions (a low barrier to entry platform widely used by the community and with a free tier), AWS Lambda (a scalable commercial cloud), and OpenWhisk (open-source FaaS deployed on both cloud and edge). This shows FaaSr is generalizable to serverless container-based systems that do not offer typical FaaS APIs (e.g. GitHub).

II. Related Work

While there is an extensive literature in workflow systems, FaaS-based workflows are a more recent development. Pegasus [9] and HTCondor [10] are flagship projects supporting DAG-based scientific workflows. However, they rely on the deployment of managed servers/VMs to host the execution of workflows. In contrast, FaaSr supports serverless platforms without any managed servers/VMs. Nextflow [11] aims at enhancing the reproducibility of computational workflows in cloud and local environments. However, unlike FaaSr, the cloud bindings in Nextflow are based on managed cloud VMs (e.g. AWS EC2 and Google Cloud).

In [2], the authors survey serverless computing for scientific applications, highlighting several of the potential benefits: intuitive abstraction of functions, facilitating programming by abstracting away low-level resource management, elastic resource management, scalability, and packaging of complex software stacks in containers. In [1], the authors overview different approaches to supporting the deployment of workflows in FaaS infrastructures, and provide quantitative experiments validating the use of the HyperFlow engine in both AWS Lambda and Google Cloud Engine for the Montage application. The FaaSr approach is motivated by similar arguments, but is novel in how it implements the "decentralized model" described in [1], thereby supporting cross-platform serverless execution without the need for a managed workflow engine server/VM. This is fundamentally different from HyperFlow, which uses a queue model that requires its own workflow engine server.

A closely-related work also motivated by the potential of cross-platform serverless computing in scientific workflows is XFaaS [6], [7]. However, our approach is different from XFaaS in several ways: 1) while XFaaS uses a dedicated cloud server as a workflow orchestrator, FaaSr does not require a dedicated cloud server running a workflow engine; 2) unlike XFaaS, FaaSr does not require cloud queues or databases—instead, FaaSr implements a novel approach using passive S3 cloud storage for coordination; 3) FaaSr integrates support for file-based I/O over S3 (including Arrow), while XFaaS uses function payloads for arguments and return values; and 4) FaaSr provides the flexibility of scheduling functions to providers at the granularity of individual functions, while XFaaS does so at the granularity of sub-graphs. QuickFaaS [12] also aims at portability across FaaS providers, but does not support cross-platform workflows. Triggerflow [13] implements trigger-based orchestration of serverless DAG workflows; however, unlike FaaSr, it requires a controller responsible for creating workflow workers in Kubernetes, a front-end RESTful API, and a database.

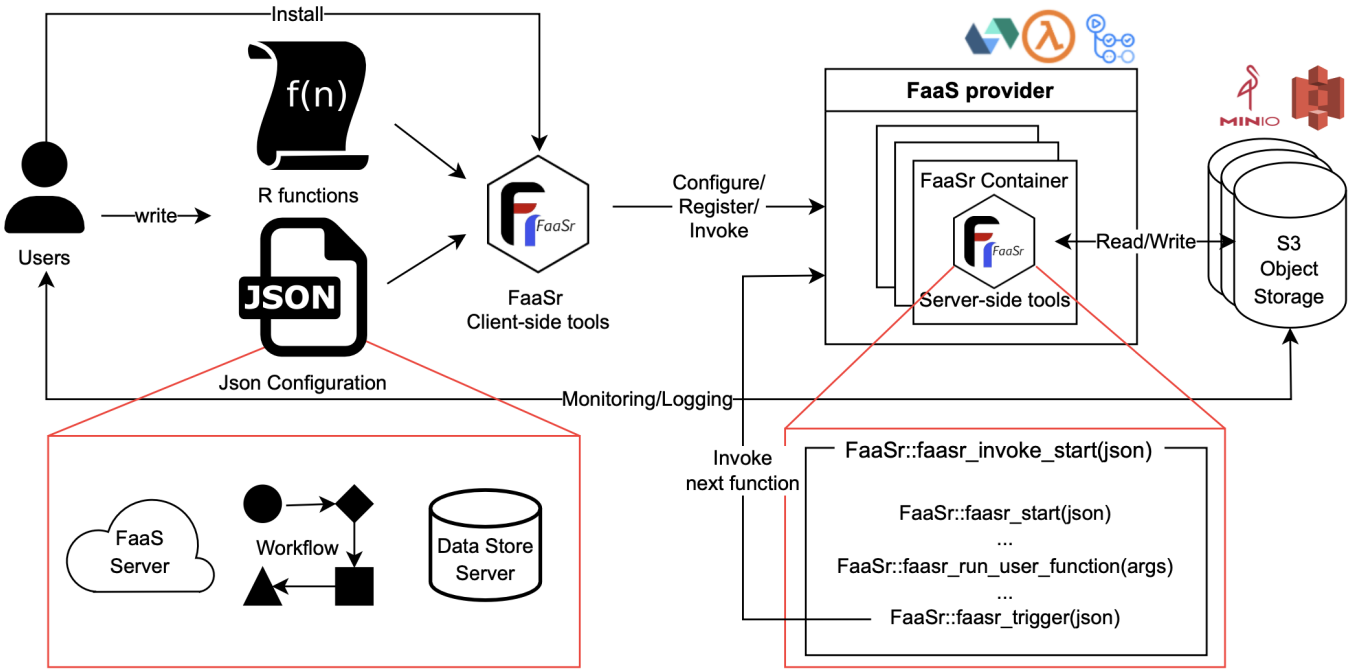

Fig. 1: Overview of FaaSr Design. *Users* write *R functions* and declare workflow *JSON configurations* in a FaaS-agnostic, abstracted fashion. *FaaSr client-side tools* generate FaaS-specific REST API calls on behalf of the user to configure, register and invoke actions. *FaaSr server-side tools* are invoked within the container runtime to access S3 files, log messages, and trigger successor DAG actions.

PyWren [14] and NumPyWren [15] also leverage serverless computing but, in contrast to FaaSr, these are specific to the AWS Lambda service and Python, and do not offer a generic framework for multiple serverless frameworks or file I/O over S3 buckets. The funcX system [5] is also Python-based, and offers a distributed FaaS platform across federated cyberinfrastructure. Unlike FaaSr, which uses cloud-native FaaS APIs (e.g. AWS Lambda), funcX primarily focuses on federated private clouds, and requires the deployment of custom endpoints to execute in public cloud VMs. PONCHO [16] is also Python-based, and focuses on dynamically packaging software dependencies for serverless Python applications. A related R package is *targets* [17]. A key difference is that FaaSr is designed to deploy event-driven workflows to serverless clouds, while *targets* focuses on increasing the efficiency of the execution of complex workflows, and does not use FaaS.

Overall, a key observation as it relates to FaaSr is that none of the existing systems surveyed support a combination of: 1) serverless coordination and a file-based abstraction using S3 object storage, 2) seamless deployment across multiple FaaS back-ends without requiring managed servers, and 3) native R-language APIs that support deployment of customizable containers/micro-VM runtimes.

III. Design

The design of FaaSr encompasses three major modules, as illustrated in Figure 1: an R package, a container runtime image, and workflow configuration structured in JavaScript Object Notation (JSON). The FaaSr R package is installable from the CRAN repository, and the container runtime layers FaaSr and its server-side "stubs" upon a baseline rocker/tidyverse image. Both CRAN and Rocker are widely used in the community, contributing to lower barriers to adoption. The FaaSr package contains both client-side tools for interactive usage at a user's desktop (e.g. to register and invoke workflows) and server-side functions that are invoked at runtime by a FaaS instance (e.g. to access files in S3). This is done such that *user code is completely platform-agnostic; all FaaS-specific API calls are implemented in FaaSr and abstracted from the user.*

The JSON-formatted configuration declares a FaaSr workflow and serves as a blueprint that can be reused, facilitating sharing. The configuration allows users to declare: 1) which FaaS and S3 services to use; 2) the names of user functions to invoke; 3) their dependencies; 4) the Git repositories from which to fetch function code; and 5) function arguments. In subsequent sections, the following terminology is used:

- **Node:** an individual vertex in the graph (DAG) representation of a FaaSr workflow.
- **Action:** refers to the run-time invocation of the workflow node by a FaaS provider, i.e., the container or micro-VM instance that encapsulates an execution environment where a user-defined R function is then executed.
- **Function:** the user-provided R function executed by an Action.

A. JSON Configuration

FaaSr exposes workflow configuration as a JSON-formatted file; the software also provides a GUI tool that allows users to compose, edit, and generate FaaSr schema-compliant JSON configurations by declaring:

- **ComputeServer:** provides FaaS-specific configurations for one or more serverless systems. Each compute server platform is described by a unique string name (e.g. MyAWS, MyOW), its type (e.g., Lambda, OpenWhisk), and endpoint information (e.g. URL, region).
- **DataStore:** provides configuration for one or more S3 storage services. Each data store is also described by a unique string name (e.g. MyAWS, MyMinio), endpoint URL, and bucket name.
- **FunctionList:** specifies action names and the user functions they invoke, their arguments, the name of the ComputeServer used for execution, and the name(s) of subsequent action(s) to invoke.

The workflow configuration is serialized as a JSON payload and transferred over HTTPS to each action, along with an invocation trigger, throughout workflow execution. In FaaSr, API payloads (which are limited in size) are only used for workflow configuration; the bulk of data is accessed through the S3 interface of a DataStore. The credentials used for ComputeServer and DataStore are stored in the user's desktop through credential environment variables that are automatically inserted by the client-side FaaSr tools upon action invocation.

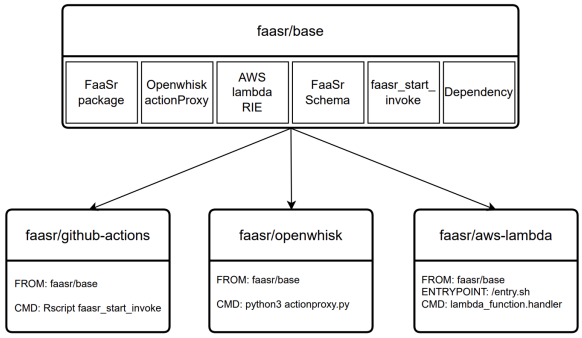

Fig. 2: Container images for each FaaS provider. The base image is built to include dependencies common to all FaaS platforms, while the platform-specific images differ with respect to container entry point and initialization script.

B. FaaSr Container

In the context of supporting R functions, FaaSr adopts containerization as its underlying technology and consolidates the R runtime environment and FaaSr package in a unified container image that can be stored in different registries (e.g. DockerHub, AWS Elastic Container Registry, and GitHub Container Registry) and deployed on multiple platforms.

The base/default FaaSr image is built starting from an R runtime environment actively maintained and widely used in the community (with over 5 million pulls): rocker/tidyverse. On top of this base image, the FaaSr package and its dependences are installed. Finally, to address the inherent heterogeneity among FaaS providers, each provider-specific container image (e.g. for GitHub Actions, OpenWhisk and AWS Lambda) is derived from the FaaSr base image. Specifically, provider-specific images are configured with distinct entry points and initialization scripts, aligning with the specific requirements of each provider (Figure 2).

While the base image works for the majority of R applications, users are empowered to craft their own custom container images. This extends the flexibility of the system, enabling advanced users to tailor the environment to their specific needs (e.g. by packaging a binary that is invoked by R) while supporting the common case with a pre-built image. Users can specify which container image(s) to use in the JSON-formatted configuration file for their functions. In cases where no explicit image is specified, FaaSr defaults to the baseline image.

C. FaaSr R Package

The FaaSr R package provides functions exposed for both client-side (the user's desktop) and server-side (container or micro-VM runtime at the FaaS provider). These functions hide the complexity associated with dealing with low-level REST API interfaces for FaaS and storage services, including: FaaS action registration, payload marshaling and parsing, DAG cycle checking, argument extraction and function invocation, action triggering, and file I/O.

1) Client-Side Functions: Client-side functions play a crucial role in simplifying user interaction with the FaaS platform. FaaSr introduces a layer of abstraction on top of the provider-specific REST API, consolidating the necessary command sets for users. These client-side functions operate within the user's desktop computer environment (e.g. the Rstudio graphical user interface) and assist in actions such as registering, invoking, and setting programmable 'cron' timers for event triggers. FaaSr invokes REST API calls using cURL. Users need to have the appropriate authorization credentials configured at their client for each FaaS and S3 provider.

2) Server-Side Functions: Server-side functions are invoked when the FaaS provider deploys the FaaSr container. *These functions allow users to write functions in R without any FaaS-specific code*. In other words, users need not worry about how functions are invoked, how to parse payloads to extract their arguments, or how to trigger downstream functions in the DAG workflow. Specifically, server-side FaaSr functions check workflow configurations for cycles; parse arguments from the JSON configuration file; invoke the user-provided R function; coordinate execution when multiple triggers are present; and print any error messages to logs. The key functions are:

FaaSr::faasr_invoke_start: is the FaaSr entry point that executes in a deployed container and implements: downloading user functions (and dependencies) from external sources (e.g. CRAN, GitHub), reading the configuration payload, and initiating the execution by calling FaaSr::faasr_start (below). Because different FaaS providers invoke this entry point in

different ways (e.g. OpenWhisk uses a custom Python "actio-proxy", whereas AWS uses a shell script) this function handles FaaS-specific invocation methods.

FaaSr::faasr_start: is the "pre-user-function" stub that implements: JSON schema validation, argument parsing, and invoking the execution of the user-provided function (see below). It verifies the workflow's correctness (i.e. no cycles nor unreachable nodes) and verifies that the S3 storage REST API endpoint can be accessed. It also determines whether to proceed or self-abort (Section III-D) *before* invoking the user function, and manages states and logs.

FaaSr::faasr_run_user_function: is invoked by FaaSr::faasr_start if an action does not self-abort. FaaSr::faasr_run_user_function reads the JSON configuration, extracts the function name and arguments, and proceeds to execute the user's R function with the provided arguments. In addition, this function handles errors generated by the user's function and issues warnings to users through logs.

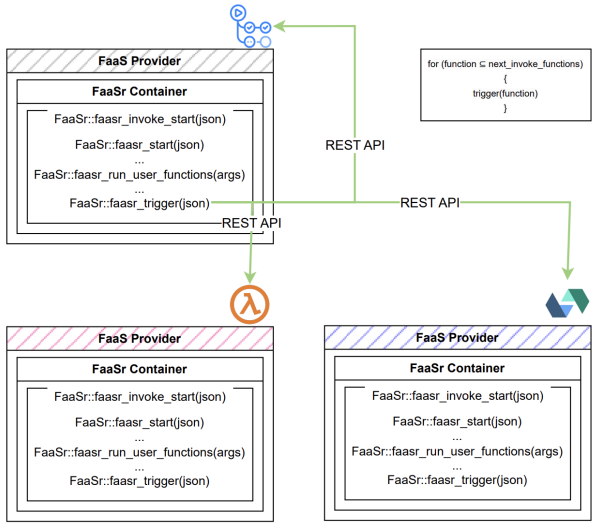

Fig. 3: FaaSr server-side triggering function. FaaSr parses the JSON configuration payload, and checks that there are no cycles in the graph. If an action has successors, they are invoked using the REST APIs of configured platform (AWS Lambda, OpenWhisk, GitHub Actions) automatically; users are not exposed to the trigger interface.

FaaSr::faasr_trigger: is the "post-user-function" stub used to trigger subsequent workflow actions, potentially on different FaaS platforms. FaaSr utilizes a JSON configuration to identify and trigger the invocation of the next set of action(s). Depending on the FaaS type associated with these actions, FaaSr generates a REST API call directed at each respective FaaS provider. These requests are singular in occurrence, non-blocking, and any potential errors are managed through the response provided by the API call. The method of triggering is depicted in Figure 3.

D. Workflows and S3-backed serverless coordination

FaaSr supports the representation of workflows as a static directed acyclic graph (with a single entry/invocation node) encoded in a JSON-formatted document. The JSON-encoded workflow is sent to the entry action as a payload using REST APIs of the FaaS provider, and actions forward the workflow representation to successor actions. Each action performs a correctness check on the workflow graph; this check verifies that nodes have unique names and uses a Depth-First Search (DFS) to ensure that the graph is cycle-free and that all nodes are reachable. If these checks fail, the action self-aborts before executing the user function, and no actions are triggered.

Because each action parses the workflow at runtime, each action can build the sets of both its predecessors and successors in the graph. These sets are used to determine: 1) which actions to trigger after executing the user function (successor set, in **faasr_trigger**), and 2) which actions to wait for completion before executing the user function (predecessor set, in **faasr_start**). The simplest (and common-case) pattern of a node in the DAG having a single predecessor and a single successor is handled by the action executing immediately after receiving a trigger, and sending a single trigger at the end of execution. A node with multiple successors is handled by the action iterating over the successor set, and sending multiple triggers, where each trigger uses the REST API of the target action. The case that requires additional complexity is when a node has multiple predecessors in the graph.

Consider a simple example where node *A* in the graph has a predecessor set *(B,C)*. At runtime, FaaSr actions corresponding to both B and C will send triggers to invoke *A*'s action. Thus, two FaaS actions will be invoked for node A. Using the representation *A.1* and *A.2* for these two actions, the approach taken by FaaSr is that one and only one of the actions (i.e. *A.1* or *A.2*) should successfully execute a user function, while the other action should self-abort—after parsing the graph, but before executing the user function. Generalizing to a predecessor set of size *N*, only a single action should execute the user function through completion, after all predecessors have completed, while the other *N-1* actions should self-abort.

FaaSr coordinates this process with the only requirement being that the actions have read/write access to S3 objects. This allows it to support the execution of individual actions across different FaaS platforms, without relying on either platform-specific workflow support (e.g. AWS Step Functions, OpenWhisk Composer) or cloud queues (e.g. AWS Simple Queue Service). The coordination works as follows.

First, each workflow run is associated with a unique identifier (e.g. a UUID), and a folder in S3 storage is created with this name. Within this folder, an action (also with a unique name, e.g. *B* or *C*) commits a file to storage with a well-defined name derived from the action (e.g. *B.done*, *C.done*) after finishing execution of the user function. This allows a successor action to self-abort if the rank of the set of *done* actions is smaller than the rank of the predecessor set. Figure 4 illustrates this scenario.

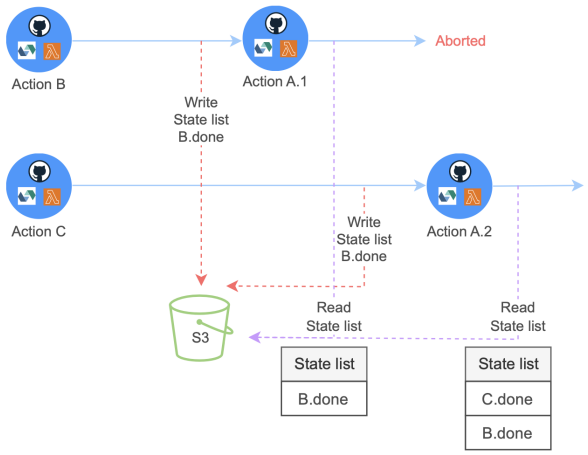

Fig. 4: Example where graph nodes *B* and *C* are predecessors of *A*. At runtime, actions *B* and *C* both invoke action *A*, leading to two FaaS action invocations *A.1* and *A.2*. In this timeline, action *A.1* is triggered by action *B*; however, action *C* has not committed *C.done* yet. As a result, action *A.1* self-aborts, while *A.2* eventually executes the user function.

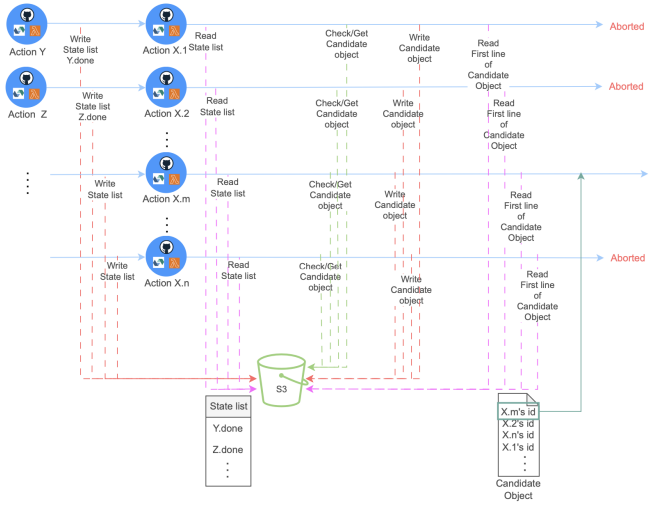

Fig. 5: If all predecessor actions are marked as *done* state, each successor action attempts to write to the *candidate* file. In this example, action *X.m*, where $1 \leq m \leq n$, adds its distinct identifier as the first line in the *candidate* file object. Only action *X.m* (the "winner" that appends the first line) proceeds with execution while the other actions (the "losers" that append to subsequent lines) self-abort.

While checking the set of *.done* file in S3 allows for early self-aborts, it is not sufficient - in general, all predecessors may write their *.done* state concurrently and trigger their respective successors. FaaSr manages coordination when all predecessors are marked *done* by using an additional *candidate* set file, also stored in S3. Actions compete to append their unique identifiers to the *candidate* set file to determine which one will execute and which ones will abort. Every action initially verifies the presence of the *candidate* set file. If the file does not exist, an action generates the *candidate* set file and attempts to append it with a single line: its unique ID. The action that successfully appends the first line of the *candidate* set self-selects as the one to execute the user function, while all remaining actions self-abort. This is illustrated in Figure 5

S3-compatible storage (e.g. AWS S3 and Minio) enforce a read-after-write consistency model. This model ensures that concurrent PUT operations from users do not result in corruption, and subsequent GET operations retrieve the most recent version of objects. However, despite this consistency model, S3 does not provide an atomic read-and-set primitive. This introduces race conditions when multiple writers append to the same object using S3 PUTs/GETs. While the lack of atomic appends does not affect the per-action *done* objects, it impacts the shared *candidate* set file, which is written by all predecessors and read by their successor.

To overcome this issue, FaaSr implements a locking mechanism on top of S3 primitives to ensure before-or-after single-writer updates to the *candidate* set, without relying on support from S3 object storage for atomic appends[1]. The strategy, inspired by Lamport's solution [18] and sketched in [19], involves the utilization of a shared $flag$ array to implement an atomic read-and-set primitive. It bootstraps a read-and-set primitive when a *single* read or write is a before-or-after operation, which is satisfied by S3. This is then used to establish a critical section to append to the *candidate* set. Then, the single action that appends the first line of the *candidate* set executes the user-provided function, while all other actions self-abort without executing the user function. To handle the (uncommon) case and prevent fate-sharing where an action fails while holding a lock, actions self-abort after a timeout if they are unable to acquire the lock.

Note that this entire process of action triggering and coordination is *abstracted away from users and implemented by the faasr_start and faasr_trigger stubs*. The end user only needs to compose a valid DAG graph, which can be done in one of two ways: 1) directly editing a JSON file to conform to the FaaSr schema, or 2) using a graphical user interface (GUI) that automatically generates a compliant JSON. While there is a cost associated with self-aborts of concurrent actions, these tend to occur early in action execution and be short compared to the time taken by typical user functions.

E. File I/O and Logging

When actions terminate, no memory or local storage is retained. FaaSr also uses cloud S3 storage not only for coordination (Subsection III-D) but also for file I/O and for logging. FaaSr simplifies user interaction with storage objects with high-level functions: :$faasr\_put\_file$, $faasr\_get\_file$, $faasr\_delete\_file$, $faasr\_get\_folder\_list$, $faasr\_arrow\_s3\_bucket$ and

[1]S3 versioned storage allows for writes to create new versions and could in principle be leveraged; however, this is not the default mode of operation and applies to an entire bucket, exposing challenges of managing versions and garbage-collection to users. Hence, FaaSr uses default, non-versioned buckets.

$faasr\_log$. These methods include error handling and the setup of S3 credentials, exposing a simple primitive to the user. Users can utilize $faasr\_put\_file$ and $faasr\_get\_file$ to upload/download files to/from storage and $faasr\_log$ to record logs to storage. Users can also access S3 data using efficient Apache Parquet columnar representation using Apache Arrow over S3.

IV. Implementation

FaaSr has been implemented in R, is hosted on a GitHub repository, and is distributed as a package [8] compliant to the requirements of CRAN (Comprehensive R Archive Network), the package repository widely used by the R community. The FaaSr package invokes REST APIs using *curl* for OpenWhisk and GitHub Actions, and *paws* for AWS Lambda and S3. On the client-side (e.g. a user's desktop running RStudio), the package offers a simple abstract interface to the user, with functions: **faasr()**, which creates an object holding the workflow configuration and credentials; **register_workflow()**, which registers workflow nodes with one or more FaaS providers; and **invoke_workflow()**, which triggers the workflow execution, either immediately, or on a timer schedule.

In addition to the main FaaSr package, additional code repositories are available to automatically build and publish custom container images (from baseline Rocker images) to DockerHub, GHCR (GitHub Container Registry), and AWS ECR (Elastic Container Registry) for OpenWhisk, GitHub Actions, and Lambda runtimes, respectively. Furthermore, a graphical user interface tool is available to create and edit workflows and generate FaaS-compliant JSON configurations.

V. Case Studies

We consider application case studies as well as micro-benchmarks in our evaluation. The case studies are from the lake ecology domain and highlight the ability of FaaSr to execute complex R functions (including invocation of a binary executable from R) in multiple FaaS platforms and S3 buckets for two common workflow patterns: "bag of task" batch (where independent jobs execute as concurrent actions) and event-driven DAG workflow invocation (representative of a forecasting use case). These exercise both the whole-file S3 put/get interface and range requests through Apache Arrow over S3 using Parquet files.

A. GLM-AED

This case study deploys concurrent actions that run several iterations of the General Lake Model - Aquatic EcoDynamics (GLM-AED) lake ecosystem model [20]. GLM-AED is an open-source process-based model widely used by the lake ecology community and is integrated with R through the GLMr package. In this use case, we generate output for a large batch of GLM-AED model simulations, where each simulation is run under different conditions. Specifically, each simulation uses a unique set of parameter values (e.g., different growth rates for lake phytoplankton) to generate a training data set for a statistical model that abstracts the ecological processes in GLM-AED (i.e., a surrogate model [21]), with the ultimate goals of improving model fidelity to lake observations and speeding computation time. Other research objectives that could be addressed using the same approach include assessing model sensitivity to changes in parameter values [22] or generating diverse training data sets for machine learning models when environmental observations are sparse [23]. The original code developed to run the batch of model simulations executed each model simulation sequentially in a user's desktop. In this case study, we took the sequential code originally developed as an R script, and converted it into an R function where parallelism is extracted by partitioning the N executions of GLM-AED (iterations of a loop) across k FaaSr actions. The process of converting the sequential code to FaaSr required modifications that were primarily limited to 1) converting Rscript into a functional format with arguments, 2) passing arguments defining the task range (i.e., the parameter values for the model simulations) to the main loop, and 3) adding a $faasr\_put\_file$ call to commit outputs to remote S3 object storage. The modifications (illustrated in the code snippets below) were limited to approximately 3.9% of the code; the runtime improvement is elaborated in the next Section.

```
glm3_assemble_surrogate_dataset <-
  function(
    start, end, output_folder,
    calibration_repo){
```

```
faasr_put_file(
  local_file=model_run_file,
  remote_folder=output_folder,
  remote_file=filename)
```

B. FLARE (Forecasting Lake and Reservoir Ecosystems)

FLARE is a system used to forecast water quality in lakes and reservoirs [24]. It has provided both context and motivation for FaaSr, as the design of FaaSr has been inspired by lessons learned from FLARE, which provides automated near-term forecasts for lakes and reservoirs in the US and abroad, including lakes from the U.S. National Ecological Observatory Network [25], Virginia [24], [26], and New Hampshire [27]. In this case study, we use the FLARE workflow for Falling Creek Reservoir (FCR), a drinking water reservoir in Virginia where FLARE has previously been used [24]

In its original design, FLARE was tightly integrated with a single serverless platform - initially with OpenWhisk [28] and later with GitHub Actions. The FaaSr-enabled FLARE, in contrast, has demonstrated the ability to seamlessly run *actions from the same workflow in any of three different FaaS platforms* (GitHub Actions, OpenWhisk, AWS Lambda), as well as demonstrating the ability to execute a workflow across edge and cloud resources.

The FLARE FaaSr workflow (illustrated in Figure 6) first executes three actions that generate the 'target' data, which are observed data converted into a standardized format for use in the model execution (inflow and meteorological data) and

data assimilation (in-situ observations from the reservoir). The three targets files are stored in an S3 bucket. Next, the targets are combined with meteorological forecasts to generate an inflow forecast that is stored in an S3 bucket. Finally, once the inflow forecast is completed, the reservoir water temperature forecast is executed with the forecast output stored on an S3 bucket. Transitioning the FLARE R package and the FCR workflow to FaaSr consisted of primarily replacing low-level $aws.s3$ R package [29] interfaces with FaaSr's interfaces for S3 and Arrow, and enabling dynamic configuration of multiple S3 servers/buckets via arguments serialized through the FaaSr JSON payloads.

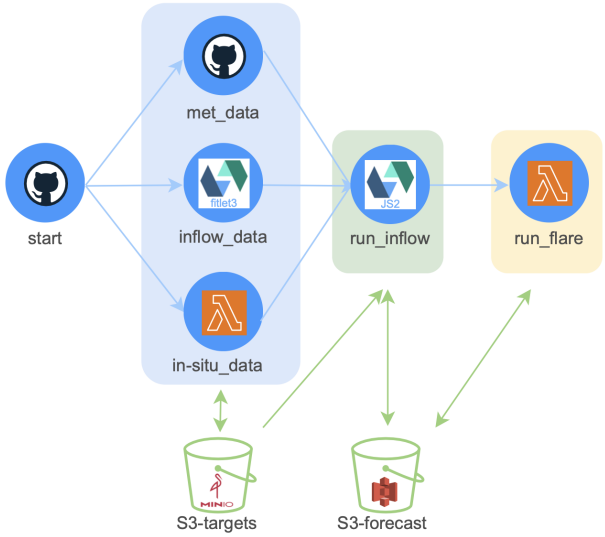

Fig. 6: FLARE forecast application workflow DAG for FCR. FaaSr supports the execution of this workflow in different FaaS platforms and different S3 buckets - the user only configures their endpoints and credentials.

While FaaSr has been primarily motivated by serverless cloud computing, its cross-platform nature generalizes to event-driven edge to cloud workflows. In ecological forecasting, such end-to-end workflows can include data preprocessing at the edge and model execution and data assimilation in the cloud. While computational capacity is limited in edge devices, a benefit of an event-driven workflow that uses the same abstractions/APIs (FaaS-invoked containers and S3 object storage) is to simplify workflow development, deployment, and reuse. To demonstrate the feasibility of this model, this case study includes an OpenWhisk cluster deployed in a low-power fitlet3 edge device and shows that, with FaaSr, *a single FLARE workflow executes across one edge and three cloud FaaS resources*, without any managed/active workflow engine.

VI. Evaluation

We have performed several experiments to evaluate the functionality and performance of FaaSr using the set of resources summarized in Table I.

TABLE I: FaaS and storage resources used in experiments

Label	Resource
OW-J	OpenWhisk, Jetstream2 [30] "medium" VM (8 core, 32GB RAM)
OW-C	OpenWhisk, CloudLab [31] cluster (48 cores, 186GB RAM)
OW-E	OpenWhisk on fitlet3 edge device (4 core, 8GB RAM)
AWS	AWS Lambda, US-east-1
GHA	GitHub Actions, free tier
S3-A	S3 service, AWS US-east-1
S3-M	S3 service, Minio on Jetstream2 "medium" VM
D-M1	Desktop: Macbook Air, 8-core M1, 8GB RAM

A. Null-function micro-benchmark

This function is designed to perform no computation, but still requires a FaaS invocation. It is used to assess FaaSr overhead, as follows (Figure 7). First, the $faasr\_start$ function is invoked upon container invocation. This acts as a stub that performs JSON payload validation, workflow DAG validation, S3 reachability check, and self-aborting for coordination (Subsection III-D). The stub then invokes the user function and, upon its return, the $faasr\_trigger$ function in the stub invokes subsequent functions, if applicable. FaaSr then wraps up by logging state and outputs to S3 and terminates. Figure 8(a) shows the latency distribution at each step for 10 runs in each of different FaaS providers. While start-up, trigger, and wrap-up latencies vary across providers, they are on the order of one or a few seconds.

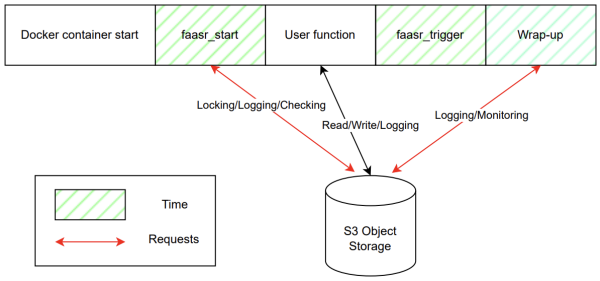

Fig. 7: Life cycle of a FaaSr action. FaaSr-specific overhead is introduced by its "stub" code surrounding user function invocation: faasr_start, faasr_trigger, and wrap-up.

We also evaluate the overhead of self-abortion on multiple invocations (Section III-D). We instrument $faasr\_start$ and vary the number of predecessors from 2 to 64. We deploy our own FaaS cluster and S3 storage (OW-C, S3-M) to create a controlled experimental setup. Containers were invoked simultaneously, and each action competes to acquire the flag and lock. As shown in Figure 8(b), while there is a variance in $faasr\_start$ times, the increase is not linear with the number of predecessors, and the latency is again on the order of a few seconds for up to 64 predecessors. Lock contention is mitigated by using exponential back-off.

Triggering successors introduces overhead because R serializes the $faasr\_trigger$ function, where each step involves data processing, formatting, and a REST API call. Results (Figure 8(c)) show that the time to invoke all triggers increases

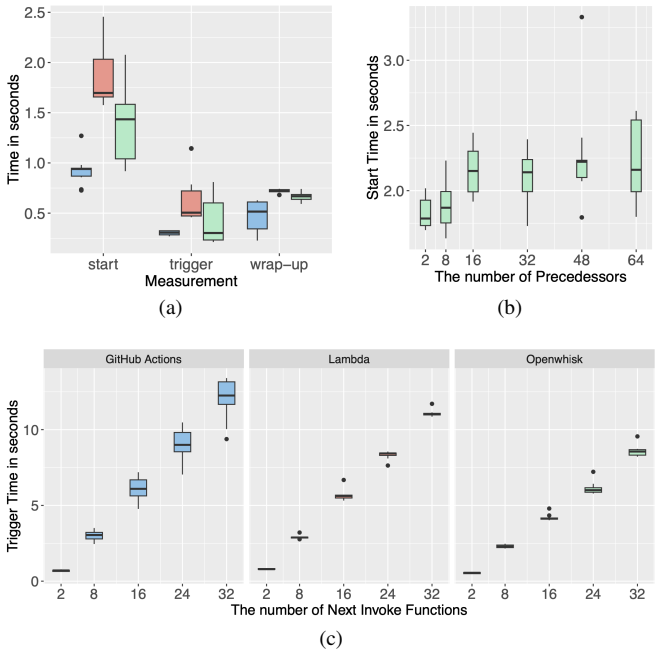

Fig. 8: Null function. (a) Overhead of each FaaSr stage; (b) start time, multiple predecessors; (c) trigger time, multiple successors. Resources: GHA (blue), AWS (red), OW-C (green)

linearly with the number of triggers. Note that while the sending of triggers by a predecessor is serialized, successors execute concurrently once triggered. Overall, these results show acceptable overheads for the target FaaSr use case scenarios, where functions take seconds to minutes to execute and where the primary goal is not raw performance, but rather to reduce barriers to entry.

B. Arrow micro-benchmark

This experiment demonstrates how FaaSr enables partial S3 object transfers by leveraging Apache Arrow [32] and Parquet [33] files on S3. For example, the FLARE case study uses the Arrow package, which exposes easy-to-use APIs in R that improve performance by: 1) splitting large datasets across multiple Parquet files with key/value pairs establishing a hierarchical folder namespace, and 2) storing each Parquet file in an efficient column-oriented format with metadata to allow byte-range transfers. The experiment uses a public dataset of taxi trip record data [34] (158 Parquet files, one per month, with total dataset size of 70GB) and the example R query [35] to find the median tip percentage for rides with fares greater than $100 in a time period. We measured the total traffic between container and S3 server using tcpdump, and logged the total number of bytes transferred during the query, repeated 20 times. For per-year and per-month queries, the average data transfer size was 480MB and 41.8MB, respectively, or approximately 0.6% and 0.06% of the 70GB dataset. A per-month query only transferred 11.1% of a single Parquet file.

C. GLM-AED application

In this experiment, we demonstrate the ability of FaaSr to seamlessly distribute a "bag of tasks" consisting of independent GLM-AED model runs across three different serverless platforms (OW-J, AWS, and GHA, Table I). Table II summarizes the experimental results, both for the execution of a single action in each platform, and the execution of 40 actions across the platforms (20 in GHA, 10 in OW-J, and 10 in AWS) as well as in a local desktop (D-M1). While each platform exhibits different execution times due to differences in hardware resources, the key conclusion is that FaaSr allows the workflow to scale out (compared to a local platform) *and* provides the user with a choice of which FaaS platform(s) to use, without the need for any FaaS-specific code.

TABLE II: GLM-AED case study: execution time of single action in each platform, and for 40 actions (concurrently in FaaSr, sequentially in D-M1)

Task	Resource	Start	Execution	Total Time
Single action	OW-J	4s	7m 26s	7m 30s
	AWS	11s	13m 28s	13m 39s
	GHA	1m 45s	7m 51s	9m 36s
	D-M1.	-	7m 14s	7m 14s
40 actions	OW-J+AWS+GHA	7s	15m 36s	15m 43s
	D-M1	-	294m 52s	294m 52s

D. FLARE application

The FLARE FCR workflow (Figure 6) successfully executes across multiple FaaS and S3 platforms. Specifically, referring to Table I, the cross-platform workflow is deployed across GHA for *start* and *met_data*, OW-E for *inflow_data*, OW-J for *run_inflow*, and AWS for both *in-situ_data* and *run_flare*, with buckets in both S3-A and S3-M. Experimental results (shown in Table III) show that, while each platform showcases unique characteristics (for example, GHA lacks warm container startup, resulting in differing invocation times), users can seamlessly deploy workflows across FaaS platforms of their choice (cloud and edge), without being exposed to low-level FaaS APIs nor vendor lock-in.

TABLE III: FLARE FCR forecast workflow execution times for each cloud FaaS platform (OW-J, AWS, GHA), and across four platforms (including an edge device, OW-E)

Action		OW-J	AWS	GHA	OW-J+OW-E +AWS+GHA
met-data	Start	6s	8s	1m 48s	1m 45s
	Exec	49s	9m 30s	1m 9s	1m 25s
in-situ-data	Start	6s	9s	1m 47s	9s
	Exec	46s	1m 1s	46s	59s
inflow-data	Start	7s	10s	1m 45s	45s
	Exec	10s	14s	12s	32s
run-inflow	Start	6s	8s	50s	5s
	Exec	6s	9s	9s	6s
run-flare	Start	6s	9s	1m 45s	9s
	Exec	6m 49s	11m 9s	10m 52s	11m 36s
Total	-	8m 2s	21m 13s	16m 33s	15m 6s

VII. Conclusions

This paper describes the design and implementation of FaaSr, a novel middleware that reduces barriers to entry for users to leverage serverless computing for scientific workflows. A key novel aspect is its ability to support cross-FaaS workflow execution using passive S3 storage for coordination rather than a dedicated workflow engine, thus avoiding vendor lock-in. Micro-benchmark results show that the performance overhead is acceptable for use case scenarios where functions take on the order of seconds to minutes to execute, and that partial S3 data transfers are possible using Apache Arrow and Parquet. Case studies based on realistic lake ecology applications show that FaaSr workflows can be successfully deployed across both edge and cloud resources—without the need for FaaS-specific user code nor managed workflow engines or servers. This allows the use of a consistent serverless abstraction of event-driven container deployment for workflows, both reducing barriers to entry and fostering workflow reuse.

References

[1] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020.

[2] M. Malawski and B. Balis, "Serverless computing for scientific applications," *IEEE Internet Computing*, vol. 26, no. 4, pp. 53–58, 2022.

[3] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2020.

[4] K. Chard and I. Foster, "Serverless science for simple, scalable, and shareable scholarship," in *2019 15th International Conference on eScience (eScience)*. IEEE, 2019, pp. 432–438.

[5] 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," in *Proceedings of the 29th International symposium on high-performance parallel and distributed computing*, 2020, pp. 65–76.

[6] A. Khochare, Y. Simmhan, S. Mehta, and A. Agarwal, "Toward scientific workflows in a serverless world," in *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, 2022, pp. 399–400.

[7] A. Khochare, T. Khare, V. Kulkarni, and Y. Simmhan, "Xfaas: Cross-platform orchestration of faas workflows on hybrid clouds," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2023, pp. 498–512.

[8] R. J. Figueiredo, S. Park, and R. Q. Thomas, *FaaSr: FaaS (Function as a Service) Package*, 2024, r package version 1.1.2. [Online]. Available: https://CRAN.R-project.org/package=FaaSr

[9] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[10] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience," *Concurrency and computation: practice and experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[11] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, pp. 316–319, 2017.

[12] P. Rodrigues, F. Freitas, and J. Simão, "Quickfaas: Providing portability and interoperability between faas platforms," *Future Internet*, vol. 14, no. 12, 2022.

[13] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, vol. 124, pp. 215–229, 2021.

[14] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 symposium on cloud computing*, 2017, pp. 445–451.

[15] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[16] B. Sly-Delgado, N. Locascio, D. Simonetti, B. Wiseman, B. Tovar, and D. Thain, "Poncho: Dynamic package synthesis for distributed and serverless python applications," in *Proceedings of the 2nd Workshop on High Performance Serverless Computing*, 2022, pp. 8–14.

[17] W. M. Landau, "The targets r package: a dynamic make-like function-oriented pipeline toolkit for reproducibility and high-performance computing," *Journal of Open Source Software*, vol. 6, no. 57, p. 2959, 2021. [Online]. Available: https://doi.org/10.21105/joss.02959

[18] L. Lamport, "A new solution of dijkstra's concurrent programming problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 171–178.

[19] J. H. Saltzer and M. F. Kaashoek, *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[20] M. R. Hipsey, L. C. Bruce, C. Boon, B. Busch, C. C. Carey, D. P. Hamilton, P. C. Hanson, J. S. Read, E. de Sousa, M. Weber, and L. A. Winslow, "A general lake model (glm 3.0) for linking with high-frequency sensor data from the global lake ecological observatory network (gleon)," *Geoscientific Model Development*, vol. 12, no. 1, pp. 473–523, 2019.

[21] R. B. Gramacy, *Surrogates: Gaussian process modeling, design, and optimization for the applied sciences*. Chapman and Hall/CRC, 2020.

[22] R. Ladwig, P. C. Hanson, H. A. Dugan, C. C. Carey, Y. Zhang, L. Shu, C. J. Duffy, and K. M. Cobourn, "Lake thermal structure drives interannual variability in summer anoxia dynamics in a eutrophic lake over 37 years," *Hydrology and Earth System Sciences*, vol. 25, no. 2, pp. 1009–1032, 2021.

[23] A. Karpatne, X. Jia, and V. Kumar, "Knowledge-guided machine learning: Current trends and future prospects," *arXiv preprint arXiv:2403.15989*, 2024.

[24] R. Q. Thomas, R. J. Figueiredo, V. Daneshmand, B. J. Bookout, L. K. Puckett, and C. C. Carey, "A near-term iterative forecasting system successfully predicts reservoir hydrodynamics and partitions uncertainty in real time," *Water Resources Research*, vol. 56, no. 11, p. e2019WR026138, 2020.

[25] R. Q. Thomas, R. P. McClure, T. N. Moore, W. M. Woelmer, C. Boettiger, R. J. Figueiredo, H. R. T., and C. C. Carey, "Near-term forecasts of neon lakes reveal gradients of environmental predictability across the us," *Frontiers in Ecology and the Environment*, vol. 21, no. 5, June 2023.

[26] H. L. Wander, R. Q. Thomas, T. N. Moore, M. E. Lofton, A. Breef-Pilz, and C. C. Carey, "Data assimilation experiments inform monitoring needs for near-term ecological forecasts in a eutrophic reservoir," *Ecosphere*, vol. 15, no. 2, 2024.

[27] W. M. Woelmer, R. Q. Thomas, F. Olsson, B. G. Steele, K. C. Weather, and C. C. Carey, "Process-based forecasts of lake water temperature and dissolved oxygen outperform null models, with variability over time and depth," *SSRN (preprint)*.

[28] V. Daneshmand, A. Breef-Pilz, C. C. Carey, Y. Jin, Y.-J. Ku, K. C. Subratie, R. Q. Thomas, and R. J. Figueiredo, "Edge-to-cloud virtualized cyberinfrastructure for near real-time water quality forecasting in lakes and reservoirs," in *2021 IEEE 17th International Conference on eScience (eScience)*. IEEE, 2021, pp. 138–148.

[29] T. J. Leeper, *aws.s3: AWS S3 Client Package*, 2020, r package version 0.3.21.

[30] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, and J. Towns, "Access: Advancing innovation: Nsf's advanced cyberinfrastructure coordination ecosystem: Services support," in *Practice and Experience in Advanced Research Computing (PEARC '23)*, 2023.

[31] D. Duplyakin, R. Ricci, R. Aleksander Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14.

[32] N. Richardson, I. Cook, N. Crane, D. Dunnington, R. François, J. Keane, D. Moldovan-Grünfeld, J. Ooms, and Apache Arrow, *arrow: Integration to 'Apache' 'Arrow'*, 2023, r package version 13.0.0.1. [Online]. Available: https://CRAN.R-project.org/package=arrow

[33] A. S. Foundation. Apache parquet. [Online], Available: https://parquet.apache.org/. [Accessed: Mar-2024].

[34] N. Taxi and L. Commission. Tlc trip record data. [Online], Available: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. [Accessed: Mar-2024].

[35] A. S. Foundation. Arrow: Working with multi-file data sets. [Online], Available: https://arrow.apache.org/docs/r/articles/dataset.html. [Accessed: Mar-2024].