DAGit: A Platform For Enabling Serverless Applications

Anubhav Jana, Purushottam Kulkarni and Umesh Bellur
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{anubhavjana, puru, umesh}@cse.iitb.ac.in

***Abstract*—Serverless computing is rapidly gaining popularity for provisioning composable, auto-scalable and cost-effective applications. An important mechanism for deploying serverless applications is specification of function workflows (via DAGs). The end-to-end life cycle of this process being DAG specification, DAG orchestration, execution of DAG components and persistent storage of application outputs. To the best of our knowledge, an open-source platform that offers functionality along all these components does not exist. Towards this, our primary contribution is *DAGit*, an open-source solution for serverless applications-as-a-service. The main features of DAGit are interfaces and specifications to register serverless functions, applications (via DAGs) and triggers to instantiate the serverless applications. DAGit provides a rich set of DAG primitives to enable a varied set of applications and also implements a scalable orchestrator for application execution. As part of this work, we present the architecture and design details of DAGit, and demonstrate its feature set via showcasing the specification and execution of a varied set of serverless applications. Further, we also present a performance and resource costs characterization of executing applications on the DAGit platform.**

Keywords— FaaS, serverless workflows, agile composition

I. Introduction

The function-as-a-service (FaaS) [1] model of serverless computing is rapidly gaining popularity and several popular available solutions and platforms are already available, e.g., AWS Lambda [2], IBM Cloud Functions [3], Microsoft Azure Functions [4], Google Cloud Functions [5], Apache OpenWhisk [6], Knative [7], OpenLambda [8] etc. A key component of FaaS based cloud services is the decoupling of the functionality and the setup and orchestration for executing the associated *function*. Developers register functions (via source code and/or services withing containers) and associate them with triggers for event-based execution.

While the FaaS model provides on-demand functionality service for applications, another mode of using the Faas model is to compose complete applications as a workflow, the elements of the workflow being serverless functions. Figure 1 and Figure 2 show two different example applications composed as a workflow of serverless functions. **AMBER Alert** [9] [10] is an application, to processes traffic camera input and video feeds, decode them into frames, pre-process each image, followed by using ML inference model for object detection, and further refine the inference to detect faces or cars. Similarly, **Montage** [11] is a scientific workflow application which processes a set of input images from astronomic sky surveys

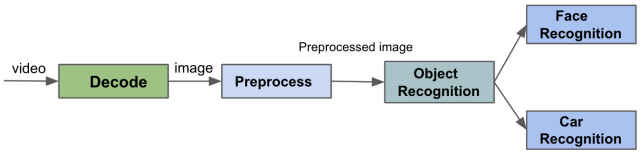

Fig. 1: The Amber Alert application as a workflow.

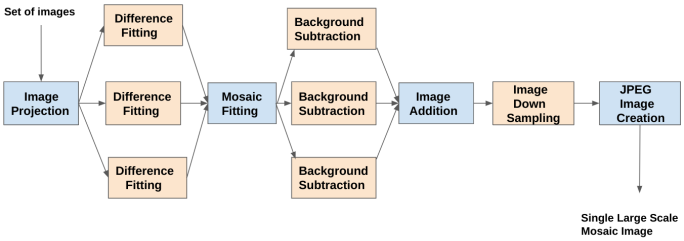

Fig. 2: The Montage application as a workflow.

and outputs a single large scale mosaic image. The application logic is embedded in a series of functions—image projection, image background subtraction, and mosaic fitting. These functions work together to re-project, align, and combine input images, and to generate metadata including information such as image location, size, and orientation and preview images. As can be seen, even from these two examples, applications as workflows involve linear ordering, one-to-many and many-to-one data dependencies, conditional branches etc.

Specifying applications as serverless workflows (as directed graphs) is attractive not only because of the flexibility of of choosing functions for composition, but also due to the agility of updating and changing application logic. Not only a new/different function implementation can be used to adapt a workflow, workflows themselves can be updated via changes to application composition specifications. The end-to-end life cycle of this process being workflow specification, workflow orchestration and data handling across workflow components, execution of serverless functions (workflow components), failure handling and persistent storage of outputs.

An important aspect of enabling the applications as serverless workflows is a service model that will allow developers and users to simultaneously contribute to function repositories and for users to compose applications using the expanding set of functions. While several workflow engines like Apache

Airflow [12] Nextflow [13]can be used to compose and schedule workflows, they are not designed for serverless platforms. While both Airflow and Nextflow have extensions to integrate with container management platforms like Kubernetes, they lack functionality to support to end-to-end life cycle of serverless applications as a service. For example, function registration, function selection, output handling are not supported as first-class features.

Further, managing the data flow between function instances of workflow has multiple design options—remote storage guaranteeing persistence and network access at the cost of higher latency or near/local storage for caching intermediate data for low latency access, but no guarantees on persistence. To summarize, existing open-source frameworks lack comprehensive support — support only a partial set of primitives or require embedded logic within functions for workflow sequencing and explicit data path handling, limiting their reusability across workflows.

As part of this work, we aim to overcome the above mentioned limitations and build a platform from the ground up to provide all functionalities of the service life cycle. The main contributions of our work are:

(i) design and implementation of DAGit, an open source solution integrated with Apache Openwhisk, for serverless workflow applications.
(ii) a platform to support a rich set of DAG primitives — linear, parallel, merge, conditional, multi-stage output. propagation, to support varied set of application requirements
(iii) an interface to query DAG metadata and corresponding individual function results.
(iv) specification to register functions, capabilities to list and select functions and specification for workflow registration.
(v) a demonstration of the comprehensive feature set and capabilities of DAGit.

The subsequent sections of this paper are organized as follows. Section II presents the Background and Related Work, providing a comprehensive overview of the relevant background information and discussing the existing work in the field. Section III describes the Requirements, outlining the specific requirements that are addressed by the proposed system. Following that, Section IV offers a detailed account of the Design and Implementation Details of DAGit, explaining the architecture and technical aspects of the system. In Section V, the Experimental Evaluation of DAGit is presented, including the methodology and results of the experiments conducted to assess its performance and effectiveness. Finally, Section VI concludes the paper by summarizing the key findings and contributions of the study, and presents potential directions for future work.

II. Background and Related Work

As the serverless computing paradigm has matured, it has become evident that building complex applications requires the coordination of multiple functions in a specific order to accomplish a particular task, and necessitating a serverless workflow orchestrator.

Apache Openwhisk Composer [14] is one such orchestrator tool to compose functions using flow primitives like linear, parallel, merge, conditional etc. A major drawback of Composer is that the **conditional** "if-else" construct expects the functions to return either a True or False value to decide conditional path. As a result, the condition to be applied on the flow has to be embedded within the function. This tight coupling of the conditional flow logic within function implementation has two main drawbacks—the function cannot be easily reused in other flows and a change in condition necessitates an update of the function implementation. Further, the Composer does not support non-binary conditions for workflow coordination and the *equal to* condition. An additional limitation with Composer is that it does not allow for generalized data dependencies across functions in a flow. All input and output of data is assumed to be linear from the current function to the next. Data dependency patterns that truly mimic a DAG (direct-acyclic graph) are not possible with Composer. In order to pass data from the first function to say the last function in a workflow, data has to be propagated through every function in the sequence of the workflow.

The main limitations of Composer include the tight coupling of conditional flow logic within function implementation, limited support for non-binary conditions and equal-to condition, lack of generalized data dependencies across functions, the inability to create data dependency patterns resembling a DAG, the requirement of Redis support for parallel execution, and the restriction of a 1:1 relationship between triggers and workflows. These limitations can make it challenging for end users to reuse functions, handle changing conditions, and manage multiple workflow triggers.

AWS Step Functions [15] is an Amazon Web Services managed service to compose and create workflows. Serverless workflows with AWS Step functions are par-per-use and are restricted to the AWS ecosystem. Migrating workflows to another cloud provider or on-premises environments require significant effort and are not straightforward. DAGit simplifies the effort by providing a clear abstraction to users, limiting the entire execution steps to a single trigger invocation. Moreover, DAGit is an open-source and cloud-agnostic platform, thus eliminating the need to have a cloud account.

Apache Airflow [16] is a workflow engine that programmatically enables the creation, scheduling and monitoring of workflows. With Airflow, data passing within functions needs to be explicitly defined in the functions itself, restricting the generality of their usage. Moreover, maximum size of intermediate data that can be stored is 48 KB and incapable of handling requirements of applications with larger intermediate Another workflow management platform, **NextFlow** [17], is a bio-informatics workflow manager that enables the development of portable and reproducible workflows. **Kubeflow** [18] is a popular open-source platform for managing and deploying machine learning workflows on Kubernetes. None of the above mentioned workflow management and orchestration services

Feature	Airflow [12]	Nextflow [13]	Synapse [21]	DAGit
Composability	✓	✓	✗	✓
Conditional Operations	✓	✓	✓	✓
Open Source	✓	✓	✓	✓
Cloud Agnostic	✓	✗	✗	✓
Data Passing Optimization	✗	✗	✗	✓

Table I: Feature-wise comparison of DAGit with state-of-the-art frameworks.

(Airflow, Nextflow, Kubeflow) provide features for end-to-end life cycle of serverless applications as a service. The features of function registration, generalized function selection and output handling are not supported as first-class features. DAGit aims to address these aspects and provide an unified tools for all aspects and requirements of a serverless workflow application setup, execution and monitoring.

Sonic [19], is a data management solution for function chains that optimizes application performance and cost, by transparently selecting the between different storage for intermediate storage — remote storage, VM-storage or direct passing, for each edge of the serverless workflow DAG.

Orion [20] is a solution to minimize to meet probabilistic guarantees for end-to-end latency of serverless workflows. The techniques use right-sizing (model based resource allocation for functions), bundling (co-location of functions) and just in time pre-warming of VM-based function runtimes.

While Sonic and Orion are not solutions for end-to-end registering, managing and hosting of serverless workflows, they solve important problems related to performance guarantees and resource provisioning, and are complementary to aims of DAGit. In fact the DAGit prototype can provide performance guarantees on workflows based on borrowing ideas from Sonic and Orion.

Towards comparing DAGit with the state-of-the-art frameworks—Airflow [12], Nextflow [13] and Synapse [21], note that the following features of DAGit are not supported by these frameworks—customizable DAG specifications, Trigger and Function registrations, Function listings, Multi-stage output propagation, 1:M trigger to workflow association and Workflow query interface.

Further, Table I shows the comparison of some common features. DAGit supports multiple function output propagation with explicit data path specification. It offers a conditional primitive with a wider range of relational operators, enabling users to define the comparison key and target directly in the DAG specification. This promotes re-usability of functions across workflows. Additionally, DAGit allows for a 1:M relationship between triggers and workflows, enabling the triggering of multiple workflows with a single trigger.

III. Requirements

The main contributions of DAGit are to enable FaaS-based service providers and end users to create custom DAG specifications, add functions and triggers to the resource pool, and provide telemetry for operations and usage. Towards meeting these goals, DAGit is built to address the following requirements—

Customizable DAG Specification: Users should have the ability to define their own DAG specifications using a json format, allowing them to specify the control and data paths of their workflows as per their requirements.

Function Integration and Customizable Trigger Specification: DAGit should provide a mechanism for users to register their own functions and triggers, enabling seamless integration into the workflow ecosystem.

Multi-Output Propagation: Users should be able to propagate data efficiently across multiple paths within a workflow, allowing for the handling and routing of data from multiple function outputs.

Conditional Operations: DAGit should support a variety of relational operators to enable the execution of conditional operations within workflows, providing flexibility in workflow design and execution. This includes the ability to specify conditions directly in the DAG specification itself, allowing for more decoupled and flexible workflows without relying on hard-coded logic within individual functions.

One-to-Many Relationship between Triggers and Workflows: The platform should support triggering multiple workflows with a single trigger event, facilitating the execution of complex workflows and enabling event-driven architectures.

Workflow Management and Monitoring: DAGit should offer comprehensive functionality for efficient workflow management and monitoring. This includes the ability to list available functions, query workflows based on their DAG ID, and individually query function IDs. These features enable users to easily navigate and monitor their workflows, enhancing overall workflow management capabilities.

By addressing these requirements, DAGit aims to provide a comprehensive unified and user-friendly open-source platform for managing and executing complex workflows, empowering users to build scalable and efficient serverless applications.

IV. Design and Implementation

In this section, we first describe the architecture and components of DAGit, then the DAGit operations setup followed by design of the DAG JSON specification and workflow trigger specification.

A. DAGit Architecture

Figure 3 illustrates the architecture and components of DAGit. The implementation of DAGit uses Apache OpenWhisk [6] for the serverless operations pipeline — trigger processing, dispatch of messages, invocation of runtimes for requests etc. The saffron-colored boxes represent OpenWhisk components, while the blue-colored box represents DAGit, which is integrated with OpenWhisk.

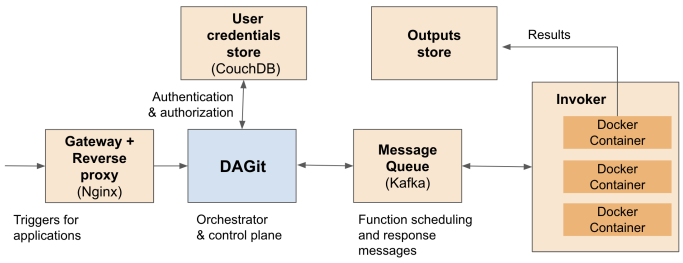

Fig. 3: DAGit integration with Openwhisk.

As shown in the figure, every trigger request first hit the **API Gateway** [22] which routes the requests to the back-end service. This also acts as a reverse proxy for the request and terminating SSL. Before processing the requests, authentication is performed using **CouchDB** [23], an open-source JSON data store. DAGit acts as an orchestrator, handling authenticated trigger requests and performing the orchestration of functions in the workflow based on the control and data flow. The orchestrator of DAGit manages and controls all incoming trigger requests for the DAG, queuing them in the **Kafka** [24] queue controlled by OpenWhisk. The individual requests for execution of functions in the given DAG is queued up in an internal queue data structure, maintained by DAGit. The Invoker consumes the trigger requests from the Kafka queue and initiates container runtimes to execute either the DAG or an individual function. The response, along with a unique DAG ID or function ID, is returned to the users and stores the outputs in an output store.

B. DAGit Operations Setup

DAGit serves as a versatile platform catering to three main categories of actions: ***DAG Registration***, ***Function Registration***, ***Trigger Registration***

We now list the requirements that DAGit expects from the variety of user it supports -

- A JSON DAG specification file: This file is used by DAG Registrars to define the structure of a Directed Acyclic Graph (DAG). It specifies the functions involved in the DAG, as well as their control sequence and data path. The DAG specification outlines the dependencies and relationships between the functions, allowing for the execution of complex workflows.
- Function source code, Dockerfile, and requirements file: These items are required by Function Registrars for deploying functions. The function source code contains the implementation logic of a specific function. The Dockerfile is used to build a Docker image that includes all the dependencies and packages required by the function. The requirements file lists the specific Python dependencies needed by the function, allowing for easy installation and management.
- JSON trigger specification: This specification defines the trigger settings for executing functions or DAGs. It includes the trigger name, the type of trigger (either a function or a DAG), and the specific function or DAG to execute when the trigger is activated. The JSON trigger specification supports a 1:M (one-to-many) relationship, allowing multiple DAGs or functions to be associated with a single trigger. This enables flexible and versatile trigger-based execution of different workflows or functions based on specific events or conditions.

By accommodating these different user categories, DAGit offers a comprehensive and collaborative environment where DAGs, functions, and triggers can be registered, orchestrated, and executed seamlessly.

C. DAG JSON Specification

We now list the required fields in a DAG specification file

- **name** : Name of the DAG
- **node_ id** : Name of the function
- **node_ id_ label**: Name to specify purpose of the function
- **primitive**: Type of primitive the action supports - condition, parallel, serial.
- **condition**: Required if primitive type is *condition*. Specifies the *source*, *operator*, and *target* fields. Otherwise this field should be left as "".
- **source**: Specifies one of the response keys of the current node_ id. Used in conditions to compare values. For e.g. if one of the keys in response json is *result*, and we want to provide a condition that if result==*even*, then specify *source* as result and *target* as even.
- **operator**: Mathematical operations like *equals*, *greater_ than*, *less_ than*, *greater_ than_ equals*, *less_ than_ equals* are accepted.
- **target**: Specify the target value. It can accept both integer and string.
- **next**: Specify the name of next node_ id to be executed. If *primitive* is specified as parallel, *next* will take list of node_ ids, else it will accept a single node_ id in "" format. If this is the last node_ id (ending node of the workflow), the field should be kept as "".
- **branch_ 1**: Specify the node_ id if primitive is specified as *condition* else should be kept as "". This is the target branch which will be executed if condition is true.
- **branch_ 2**: Specify the node_ id if primitive is specified as *condition* should be kept as "". This is the alternate branch which will be executed if condition is false.
- **arguments**: This field should be kept as blank for each node_ id. It will get auto-populated with json payload when the DAG is instantiated with the trigger.
- **outputs_ from**: Specify the list of node_ id/node_ ids (functions) whose output current node_ id (function) needs to consume (data path). For the first function in the workflow DAG, this field should be left blank. This is because the first function receives its input directly from the trigger request payload sent by clients.

Listing 1 demonstrates the usage of a JSON-based DAG representation and to define conditions and specify functions to execute in parallel using the parallel primitive. The function ***odd-even-check*** does an odd-even check on a given input and generates a json response containing a key called *result*. This

Listing 1: Example specification for conditional and parallel primitives.

```
{ "name": "odd-even-test",
"dag": [
{
   "node_id": "odd-even-check",
   "properties":
   {
      "label": "Odd Even Action",
      "primitive": "condition",
      "condition":
      {
         "source":"result",
         "operator":"equals",
         "target":"even"

      },
      "next": "",
      "branch_1": "even-print-action",
      "branch_2": "odd-print-action",
      "arguments": {},
      "outputs_from":[]
   }
},
{
   "node_id": "even-print-action",
   "properties":
   {

      "label": "Even Print Action",
      "primitive": "parallel",
      "condition": {},
      "next": ["increment-action","multiply-action
          "],
      "branch_1": "",
      "branch_2": "",
      "arguments":{},
      "outputs_from":["odd-even-action"]

   }
} ] }
```

Listing 2: A DAG registration example.

```
# Code to register a DAG
# Usage: python3 dag_register.py dag.json
import requests
import sys
import json
def server():
    url = "http://<ip>:<port>/register/dag"
    input_json_file = open(sys.argv[1])
    params = json.load(input_json_file)
    reply = requests.post(url = url,json = params,
        verify = False)
    print(reply.json())
def main():
    server()
if __name__=="__main__":
    main()
```

Listing 3: A function registration example.

```
import requests
import sys
import json
def server():
    url = "http://<ip>:<port>/register/function/
        image-bilateral-filter"
    files = [
    ('pythonfile', open(sys.argv[1],'rb')),
    ('dockerfile', open(sys.argv[2],'rb')),
    ('requirements.txt', open(sys.argv[3],'rb'))
   ]
    reply = requests.post(url = url,files = files,
        verify=False)
   print(reply.json())
def main():
    server()
if __name__=="__main__":
    main()
```

key holds the value *even* or *odd* based on the computation of the given input. This response key *result* is used as a condition within the DAG, i.e. if *result* (source) is equal to *even* (target), then control will go to the *even-print-action* (target branch) else *odd-print-action* (alternate branch).

The function ***even-print-action*** has a one-to-many relationship with the functions ***increment-action*** (which increments a given input by some factor) and ***multiply-action*** (which scales a given input by a certain factor), both of which are executed in parallel.

Listing 2 provides an example of registering a DAG to the DAGit DAG store, which is built on MongoDB [25]. The provided code snippet performs a server-side operation which reads an input DAG JSON specification, sends its content as a JSON payload in an HTTP POST request to a the DAGit API endpoint for DAG registration, and prints the response received from the server to the DAG registrar.

D. Trigger JSON Specification

We now list the required fields for trigger specification.

- **trigger_ name** : Name of the trigger
- **type** : The trigger type can be either for a DAG or a function.
- **dags** : If *type* is specified as **dag**, this field expects a list of dags else this field should be left as "".
- **functions** : If *type* is specified as **function**, this field expects a list of functions else this field should be left as "".

An example for a trigger specification is mentioned in **Listing 5** in Section V.

E. Function Registration

Listing 3 provides a sample code snippet that demonstrates the registration of a function named ***image-bilateral-filter*** using DAGit. The function registration process involves providing three essential components: the **script** containing the function's logic, the **Dockerfile** for building the function's image, and a **requirements** file specifying the dependencies required by the function. By registering the function with these components, DAGit enables seamless deployment and management of the *image-bilateral-filter* function within the platform.

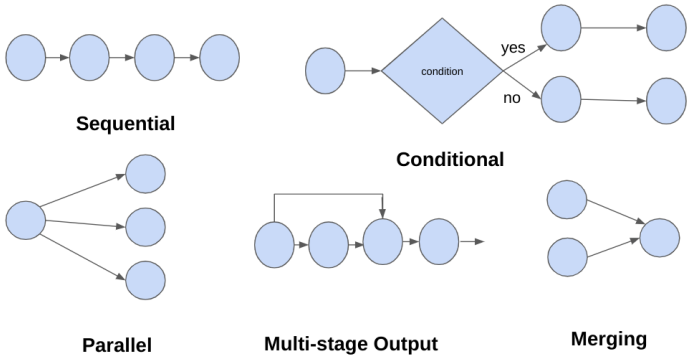

Fig. 4: Workflow primitives supported by DAGit.

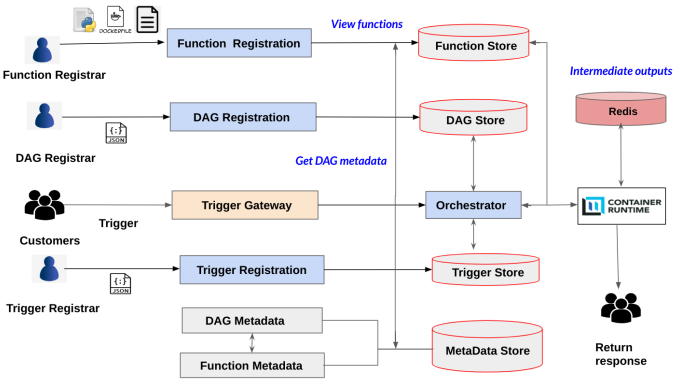

Fig. 5: Workflow operations and interactions with DAGit components.

F. Workflow Primitives

Figure 4 showcases the diverse range of primitives supported by DAGit, emphasizing its versatility and flexibility in managing and executing workflows. The **serial** primitive facilitates the sequential execution of functions, ensuring dependencies are met. The **parallel** primitive enables the concurrent execution of multiple functions without any predefined order. With the **merging** primitive, outputs from multiple functions are combined into a single output, streamlining data processing. The **conditional** primitive introduces conditional logic, allowing the execution of different branches based on specified conditions. Lastly, the **output from multiple functions** primitive empowers workflows to consume outputs from multiple functions, promoting data integration and collaboration. By offering this comprehensive set of primitives, DAGit equips users with a robust framework to design and execute workflows tailored to their specific needs.

G. Workflow Operations and Interactions

When a user provides a trigger, it is sent to the trigger gateway and authenticated by CouchDB. DAGit, acting as an orchestrator, takes charge of the authenticated trigger request. For a DAG execution request, DAGit communicates with the DAG store, searching for the specified DAG based on the user-provided DAG name. It orchestrates the execution of functions according to the control and data paths specified in the JSON of the DAG. If the request is for a single function execution, DAGit communicates with the function store and directly executes the function. To manage the workflow, DAGit's orchestrator maintains an internal queue. Prior to function execution, the function is dequeued, and the subsequent function is queued up for execution. This orchestration process ensures the proper sequence of function executions. The orchestrator of DAGit manages and controls all incoming trigger requests, queuing them in the Kafka queue controlled by OpenWhisk. The Invoker component consumes the trigger requests from the queue and initiates container runtimes to execute either the DAG or the function. The response, along with a unique DAG ID or function ID, is returned to the users. Additionally, the application outputs are stored to a cloud storage service (AWS's S3 Storage) for future reference and retrieval. Additionally, users can query DAG metadata by specifying the **DAG ID**, to get a list of function identifiers and associated metadata. Users can then individually query these **function ids** to fetch the responses from these individual functions. Table II and Table III show the schema of the DAG metadata table and function metadata table respectively.

DAG ID	Dag Name	Function IDs
3e8c7ea5	Toonify	[a1111456, c4b14cc0]
0b88dfc3	decode-blur	[b720719f, 336dd70c]

Table II: DAG metadata showing the DAG ID, DAG name and the corresponding function ids of the DAG.

Function ID	Function Response
a1111456	json-response-1
336dd70c	json-response-2

Table III: Function metadata showing the Function ID and the corresponding function output in json format.

H. Efficient Multi-Output Propagation

DAGit uses Redis as a key-value intermediate store to support the primitive of utilizing outputs from multiple functions as inputs to another function. After each completion of DAG execution, the Redis intermediate store is flushed. Redis is used an in-memory store for intermediate outputs and offers significant advantages in terms of latency reduction (compared to using remote storage options like AWS S3 as the intermediate store). By leveraging Redis, we eliminate the need for redundant data copies over the network that would have been required with AWS S3. With AWS S3, data would need to be transmitted from one function to AWS S3 and then from AWS S3 to the receiving function, resulting in substantial overhead. The Redis-based implementation optimizes latency and minimizes network data transfer overhead, contributing to efficient multi-output propagation in workflow execution.

V. Experimental Evaluation

The DAGit prototype for evaluation is composed of a cluster with a Openwhisk master node and four worker nodes. The master node is hosted on a machine equipped with an Intel Xeon E5-2683 v4 processor and 128 GB of RAM with each worker node allocated 8 CPU cores and 16 GB of RAM, and the master node allocated 10 CPU cores and 32 GB of RAM. The DAGit orchestrator executes on the master node and the

function instances on the worker nodes. Further, we enable the affinity and scheduler configurations of Openwhisk to schedule tasks across all the worker nodes that belong to the cluster.
For the invoker component, we specified certain options to customize its behavior. Specifically, (i) set the heap size available to each worker node to 512 MB. (ii) leveraged the Kubernetes container factory with a replica count of 4, benefiting from Kubernetes' container management capabilities for scalability. (iii) enabled the pod disruption budget (PDB) [26] feature to limit the number of unavailable pods during maintenance or failures. This helped maintain the overall system stability. (iv) configured to handle a maximum of 100,000 action invocations, concurrent invocations, and trigger fires per minute. Limits were also set for the maximum length of action sequences to ensure optimal performance under heavy workload.

A. Serverless Application Life Cycle with DAGit

We now show how DAGit as a platform can be used to register a DAG, register a trigger associated with the DAG, execute the application and persistently store output for clients and query DAG metadata. For this purpose, DAGit is used to deploy **Toonify**, a video analytics pipeline.

Fig. 6: Toonify Workflow Serverless Application on DAGit

As show in the Figure 6 the workflow has three functions in the pipeline — (i) the **decode** function which uses FFmpeg [27] to sample videos and generate frames, (ii) an **image filtering** function that uses median blurring, edge detection, noise removal filters and edge sharpening and masking to generate a cartoon equivalent image, (iii) the **encode** function that combines the generate images to create a cartoon video sequence. The output video is stored as a file in a S3 bucket. We first registered the DAG named **toonify** by specifying the DAG specification as shown in **Listing 4** which specifies three functions - **decode-function**, **image-bilateral-filter** and **encode-function**. The control flow is defined by specifying the next function in the workflow to execute in the *next* field, while the data path is specified in the *outputs-from* field. Following the DAG registration, the trigger is registered using the provided trigger specification, as illustrated in **Listing 5**. The specification defines the trigger name as *ToonifyTrigger*, with the ***type*** set to *dag*. The ***dags*** field specifies the name of the associated DAG as *toonify*. As it is a trigger for a DAG, the ***functions*** field is left blank.
The DAG was invoked using the specified trigger name via web request, and DAGit orchestrated the data and control flow and stored application output persistently in a S3 bucket. **Listing 6** shows the final consolidated DAG output mentioning the DAG identifier of the Toonify workflow, along with the activation and output metadata. The DAG metadata can be used to query the results using the key **dag_ id**. **Listing 7** shows the result of query using a DAG identifier that lists all

Listing 4: Toonify DAG Specification

```
{ "name": "toonify",
"dag": [
{
   "node_id": "decode-function",
   "properties":
   {
      "label": "Decode Function",
      "primitive": "serial",
      ...
   }
   },
   { ...
   },
{
   "node_id": "encode-function",
   "properties":
   { ...
     "outputs_from": ["image-bilateral-filter"]
   }
} ] }
```

Listing 5: Toonify Trigger Specification.

```
{
   "trigger_name": "ToonifyTrigger",
   "type":"dag",
   "dags": ["toonify"],
   "functions":""
}
```

the individual functions of the workflow. Function identifiers listed as part of this query result can further be used to view individual outputs of each function in the workflow execution. To estimate the overhead of the DAGit platform to perform the various orchestration actions we measured the latency for each of them. Following are the overheads for the three main tasks with the Toonify workflow— DAG registration duration was 31 ms, trigger registration duration was 26 ms and time to query the metadata store was 25 ms. Note that DAG registration and trigger registration is a single time activity, while querying is dependent on number of times the user needs feedback.
Further, the time taken to ferry data between functions (time from when a function stores data and the time when data is read by the next function was in the range of 2-3 seconds for decode function outputing 10 to 25 images.

B. Comparison of DAGit and Composer Orchestrators

DAGit not only offers a comprehensive solution for registering, executing, and querying DAG workflows and triggers but also incorporates a logging system that captures essential execution information in the dagit.log file. The logs include timestamps for function readiness, execution start, completion, and next function scheduling, along with associated function and DAG IDs. Analyzing these detailed logs enables us to estimate the **dispatch latency** overhead imposed by DAGit and the Openwhisk Composer for orchestration of workflows. Dispatch latency is defined as the time taken by the orchestrator to decide the next function(s) to execute plus the time

Listing 6: Toonify Output

```
"dag_id": "24c3de27-3b06-499d-abc4,
"result": {
   "activation_id":"b22b6d4c32fa4cddab6d4c32fa",
   "encode_output": "https://dagit-store.s3.ap-south
       -1.amazonaws.com/output.avi"
}
```

Listing 7: Toonify DAG Query Response

```
{
 "dag_metadata":
[
   {
      "dag_id": "76cc8a53-0a63-47bb-a5b5-9
          e6744f67c61",
      "dag_name": "toonify",
      "function_activation_ids:[
      "8d7df93e8f2940b8bdf93e8f2910b80f",
      "654248d5be1f4fdb8248d5be1f9fdb75",
      "6ab45fb9c9694bb0b45fb9c969bbb015"] }
]
}
```

taken to start the function(s).

In order to compare the dispatch latency and performance of execution, we used the Toonify workflow. Trigger requests to workflows instantiated via DAGit and Compose had identical inputs. The experiment was conducted for 100 iterations, where the input request payload included a video URL and the number of frames to extract from the video. The parameter for number of frames to generated as output was varied across iterations.

Table IV and Table V show the execution time of the serverless functions, the end to end latency (the response time of each workflow) and the dispatch latency (overhead of orchestrating the workflows) using DAGit and Openwhisk Composer. These statistics are reported for output requirements changing from 10 to 25 outputs frames. As can be observed from the results, the function execution times are similar between the two orchestration platforms. However, there is a notable difference in the dispatch latency, with DAGit demonstrating much lower latency (5-6 ms) compared to Openwhisk Composer (139-371 ms). This dispatch latency difference is because DAGit reads the DAG specification at the start of execution of a workflow, and while a function executes the next function is already queued for dispatch. DAGit has the next function to be executed at the head of the queue and can be fetched in O(1) time. Composer, on the other hand, spins up a separate Docker container to schedule the next function which has its own spin up (startup) time for every switch between functions.

Table VI shows the various stages of a single instance of the workflow execution. The function **decode-function** starts execution at the same time as it is dequeued. It takes just 3 ms between the completion time of decode-function and dequeing time of the next function **image-bilateral-filter**. Similarly, it takes just 2 ms for the overall control switch delay between image-bilateral-filter function and the next function

Frames	Response Time	Execution Time	Dispatch Latency
10	13.615 s	13.610 s	5 ms
15	17.370 s	17.364 s	6 ms
20	26.335 s	26.330 s	5 ms
25	35.786 s	35.781 s	5 ms

Table IV: Duration of workflow completion, function execution, and dispatch latency with DAGit and the Toonify workflow.

Frames	Response Time	Execution Time	Dispatch Latency
10	12.044 s	11.673 s	371 ms
15	18.710 s	18.568 s	142 ms
20	25.007 s	24.772 s	235 ms
25	36.449 s	36.310 s	139 ms

Table V: Durations of workflow completion, function execution, and dispatch latency with Openwhisk Composer and the Toonify workflow.

Activation ID	Duration	Execution Entity
c9b25f01ae1942	5.32 s	decode-function
c6a9bf400a314e	3 ms	**DAGit orchestrator**
24286a51cc634d	4.41 s	image-bilateral-filter
e73772ddcbb744	2 ms	**DAGit orchestrator**
b8a5db9e63304f	3.88 s	encode-function

Table VI: Toonify Workflow Log displaying the various events as part of its execution with DAGit

Activation ID	Duration	Execution Entity
2a604a28550741	340 ms	**Composer**
6971c4fe88db45	6.77 s	decode-function
c6a9bf400a314e	12 ms	**Composer**
24286a51cc634d	6.37 s	image-bilateral-filter
e73772ddcbb744	8 ms	**Composer**
b8a5db9e63304f	4.97 s	encode-function
87d58732b2ab4a	11 ms	**Composer**

Table VII: Toonify Workflow Log displaying the various events as part of its execution with Openwhisk Composer

encode-function. Additionally, there is no additional startup or stopping overhead for the DAGit orchestrator. The only overhead incurred is, thus, the dispatch latency while switching between functions in the workflow.

The total time taken to complete the workflow with DAGit is 13.62 seconds.

Table VII shows the events of execution of a single instance of Toonify workflow execution invoked via the Openwhisk Composer with identical inputs as was used in our DAGit platform. The table shows that in addition to the functions in the workflow, there is an additional overhead incurred by the container **toonify** spun by Composer which is responsible for orchestration functions in the workflow. The following latencies between events are observed— 340 ms (cold start) at startup, 12 ms and 8 ms delay (warm start) in between functions decode-function and image-bilateral-filter and image-bilateral-filter and encode function respectively and 11 ms for wrapping up the workflow. The startup latency is recorded to be in between 12-18 ms for warm container startup and 6-11 ms for stopping the container. The overhead for switching functions are from 7-14 ms. The total time taken to complete the workflow is around 18.48 seconds.

Based on this comparison we show that DAGit as (or more) efficient that Openwhisk Composer and as the length of the workflows increases we expect DAGit to yield better

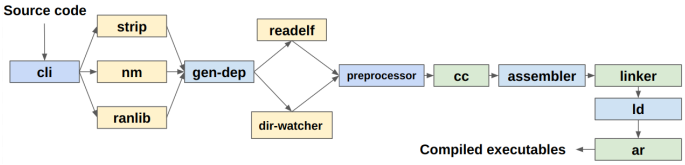

Fig. 7: Online Compiling Serverless Application on DAGit

Start	Response Time	Execution Time	Dispatch Latency	DAGit Latency %
Cold	77 s	76.98 s	20 ms	0.03
Warm	13 s	12.982 s	18 ms	0.14

Table VIII: Total latency and latency components with DAGit for the Online Compiling workflow.

performance than Apache Composer.

C. Case studies to showcase DAGit primitives

(i) Replicating a real-world serverless application

We deployed and evaluated a real-world serverless application—**Online Compiling** which is part of the ServerlessBench [28] suite of applications Figure 7) shows the application and the workflow contains a total of thirteen functions with a combination of sequential, parallel, and merge primitives.

The DAG specification snippet for this application is as shown in Listing 8. The *cli-function* is the start function followed by 3 parallel functions mentioned in the ***next*** field with ***primitive*** as ***parallel*** followed by other functions of the chain.

Table VIII provides the execution time of the serverless functions, the end to end latency (the response time of each workflow) and the dispatch latency of the Online Compiling workflow, averaged over 10 iterations. DAGit completely eliminates the startup latency as incurred in the approach followed by Openwhisk Composer and the only overhead is that of the dispatch latency which incurs a very negligible percentage of the response time (0.03 % for cold starts and 0.14 % for warm starts).

(ii) Support for Multi-Stage Output Propagation

Next we demonstrate DAGit's feature to support workflows which contain multi-stage output propagation, i.e. outputs from multiple functions as an input to a function in the workflow. Note that Apache Composer does not support this feature. We demonstrate this primitive by deploying a **Text Sentiment Analysis** [29] workflow, consisting of three functions—**fetch-sentences**, **calculate-sentiment** and **create-sentiment-report** as shown in Figure 8. The function **calculate-sentiment** takes input from the output of **fetch-sentences** and the function **create-sentiment-report** takes input from the outputs of both fetch-sentences and calculate-sentiment. Input to the function *fetch-sentences* is via the trigger, and is an URL that points to input text article. For text located at a given URL the function extracts the summary, and returns the individual sentences of the summary as a list. The next function *calculate-sentiment* takes the list of sentences from fetch-sentences, analyzes the sentiment of each sentence using TextBlob [30]. a library for processing textual data and returns a dictionary with a

Listing 8: Online Compiling DAG Specification

```
{
"name": "online_compiling",
"dag": [
{
   "node_id": "cli-function",
   "properties":
   {
      "label": "CLI Function",
      "primitive": "parallel",
      "condition":{},
      "next": ["strip-function","nm-function","
          ranlib-function"],
      ...
      "outputs_from":[]
   }
   },
   {
   "node_id": "strip-function",
   "properties":
   {
      "label": "Strip unceesaary lines Action",
      "primitive": "serial",
      "next": "generate-dep",
      "outputs_from": ["cli-function"]
   }
   },
   {...}
   {
      "node_id": "ar",
      "properties":
      {
         "primitive": "serial",
         "condition":{},
         "next": "",
         "outputs_from": ["ld"]
         }
   } ] }
```

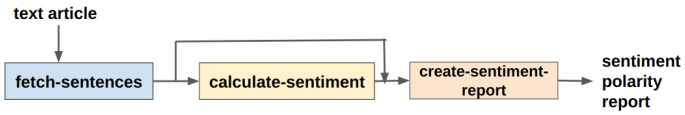

Fig. 8: The Text Sentiment Analysis application as a workflow.

list of sentiment scores for each sentence. The next function *create-sentiment-report* takes input from outputs of both fetch-sentences and calculate-sentiment, i.e., takes lists of sentences and their corresponding sentiment scores, processes them to create a tabular report, and returns the report as a formatted string. The DAG specification for the **text-sentiment-analysis** application as shown in **Listing 9**. The **outputs-from** field of the specification lists two functions that the ***create-sentiment-report*** report depends on.

The final result of the application is a report which presents the sentences and their associated sentiment scores in a structured manner allowing for easy analysis and interpretation. This sentiment polarity report provides a measure of sentiment for each sentence, ranging from -1 to 1. Positive sentiment is indicated by values greater than 0, while 0 signifies neutral sentiment, and values less than 0 represent negative sentiment. Table IX shows a output for three sentences from a sample text input.

Listing 9: Text Sentiment Analysis DAG Specification.

```
{ "name": "text-sentiment-analysis",
   "dag": [
   {
      "node_id": "fetch-sentences",
      "properties":
      {
         "label": "Fetch Sentences",
         "primitive": "serial",
         "condition":{},
         "next": "calculate-sentiment",
         ....
      }},{
     ...},
   {
   "node_id": "create-sentiment-report",
   "properties":
   { ...
      "outputs_from": ["fetch-sentences","calculate-
          sentiment"]
   }}] }
```

Sentence	Sentiment score
Although mathematics is extensively used for modeling phenomena, the fundamental truths of mathematics are independent from any scientific experimentation	0.00
Rigorous reasoning is not specific to mathematics, but, in mathematics, the standard of rigor is much higher than elsewhere	0.08
Mathematical theories and concepts often remain purely theoretical, with limited practical applications and little impact on real-world phenomena	- 0.23

Table IX: Sentiment polarity report showing sentiment corresponding to each sentence (-1 to 1).

VI. Conclusions

In this work, we designed and developed, DAGit, a platform for supporting the end-to-end life cycle of deploying serverless workflow applications. DAGit uses JSON-based specifications for registration of functions, workflows and triggers, and accompanied with a orchestrator supports a rich set of workflow primitives. We demonstrated the features and functionalities of DAGit via an experimental study with different workflows. We also showed that the overheads of DAGit or less than that of Apache Composer, while supporting a larger set of workflow primitives. DAGit will be released as a open source project and will be available for further extensions and usage.

As part of future work, we intend to extend diversity of the serverless functions store and also encode varied workflow application examples. We also plan to extend the list of supported primitives to include barrier-style synchronization, loops. Further, support for GPU-based functions, telemetry extensions, and failure handling are in the pipeline.

References

[1] E. Jonas *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," EECS Department, University of California, Berkeley, Tech. Rep., Feb 2019.

[2] "Aws lambda," https://aws.amazon.com/lambda/, retrieved July 5, 2023.

[3] "Ibm cloud functions," https://cloud.ibm.com/functions/, retrieved July 5, 2023.

[4] "Azure durable functions," https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp-inproc, retrieved July 5, 2023.

[5] "Google cloud functions," https://cloud.google.com/functions, accessed April 24, 2023.

[6] "Apache openwhisk," https://openwhisk.apache.org/, retrieved July 1, 2023.

[7] "Knative," https://knative.dev/docs/functions/deploying-functions/, accessed April 23, 2023.

[8] "Openlambda," https://www.crunchbase.com/organization/openlambda, accessed April 23, 2023.

[9] H. Shen *et al.*, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019, p. 322–337.

[10] H. Zhang *et al.*, "Live video analytics at scale with approximation and delay-tolerance," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17, 2017, p. 377–392.

[11] D. Katz *et al.*, "Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, 05 2010.

[12] "Apache airflow," https://airflow.apache.org/, accessed May 4, 2023.

[13] "Nextflow," https://www.nextflow.io/, accessed May 4, 2023.

[14] "Openwhisk compose," https://github.com/apache/openwhisk-composer/, retrieved April 20, 2023.

[15] "Aws step functions," https://aws.amazon.com/step-functions/, retrieved July 5, 2023.

[16] "Apache airflow," https://github.com/apache/airflow, accessed May 4, 2023.

[17] "Nextflow git," https://github.com/nextflow-io/nextflow, accessed May 3, 2023.

[18] "Kubeflow," https://www.kubeflow.org/, retrieved July 5, 2023.

[19] A. Mahgoub *et al.*, "Sonic: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.

[20] M. Ashraf *et al.*, "Orion and the three rights: Sizing, bundling, and prewarming for serverless dags," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Jul 2022, pp. 303–320.

[21] "Synapse," https://github.com/serverlessworkflow/synapse/, retrieved September 29, 2023.

[22] "Nginx api gateway," https://www.nginx.com/resources/glossary/api-gateway/, retrieved July 16, 2023.

[23] "Apache couchdb," https://couchdb.apache.org/, retrieved July 16, 2023.

[24] "Apache kafka," https://kafka.apache.org/, retrieved July 16, 2023.

[25] "Mongodb," hhttps://www.mongodb.com/, retrieved July 15, 2023.

[26] "Pod disruption," https://kubernetes.io/docs/concepts/workloads/pods/disruptions/, retrieved July 16, 2023.

[27] "Ffmpeg," https://github.com/FFmpeg/FFmpeg, accessed April 27, 2023.

[28] "Serverlessbench," https://github.com/SJTU-IPADS/ServerlessBench/, retrieved September 30, 2023.

[29] "Text sentiment analysis," https://aws.amazon.com/what-is/sentiment-analysis/, retrieved July 16, 2023.

[30] "Textblob: Simplified text processing," https://textblob.readthedocs.io/en/dev/, retrieved July 16, 2023.