Appendix: Artifact Description

I. Overview of Contributions and Artifacts

A. Paper's Abstract

Scientific research increasingly relies on distributed computational resources, storage systems, networks, and instruments, ranging from HPC and cloud systems to edge devices. Event-driven architecture (EDA) benefits applications targeting distributed research infrastructures by enabling the organization, communication, processing, reliability, and security of events generated from many sources. To support the development of scientific EDA, we introduce Octopus, a hybrid, cloud-to-edge event fabric designed to link many local event producers and consumers with cloud-hosted brokers. Octopus can be scaled to meet demand, permits the deployment of highly available Triggers for automatic event processing, and enforces fine-grained access control. We identify requirements in self-driving laboratories, scientific data automation, online task scheduling, epidemic modeling, and dynamic workflow management use cases, and present results demonstrating Octopus' ability to meet those requirements. Octopus supports producing and consuming events at a rate of over 4.2 M and 9.6 M events per second, respectively, from distributed clients.

B. Paper's Main Contributions

The contributions of our work are:

- C_1 Investigation of using scientific EDA to design robust distributed applications, a survey of use cases to determine requirements, and discussion of the benefits and limitations of applying EDA to five science applications.
- C_2 Design and implementation of a hybrid, cloud-hosted, multi-user event fabric, Octopus, along with its open-sourced software ecosystem.
- C_3 Evaluation of Octopus' performance, scalability, and suitability for scientific use cases.

C. Computational Artifacts

The computational artifacts developed as part of this research include:

- A_1 Octopus SDK and a walk-through notebook: github.com/globus-labs/diaspora-event-sdk/releases/tag/v0.3.5
- A_2 Evaluation Methodology and Results: doi.org/10.5281/zenodo.10975534

The relationship between the artifacts and contributions is listed below.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_2	Figure 2 Listing 1
A_2	C_3	Tables 3 Figures 3-5, 7-8

Contributions C_1 are not directly supported by the two computational artifacts. These contributions are primarily conceptual, synthesizing lessons learned from case studies and preliminary development of Octopus, focusing on the strategic insights necessary for applying EDA in scientific contexts.

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

As a part of the Octopus software ecosystem, the Octopus SDK provides a user interface to interact with Octopus Web Service for credential, topic, and trigger management features. Included in this package are the source code of the latest SDK and a Python notebook for a guided walk-through of these features, demonstrating the implementation of the cloud-hosted event fabric and its triggers for reliable real-time event processing.

Expected Results

The notebook is organized sequentially to introduce credentials, topics, and trigger management features. Specifically, the main goal of each section is to showcase:

- **Credential Management**: the SDK retrieves and stores cluster authentication credentials from Octopus Web Service, enabling users to connect and interact with the event fabric.
- **Topic Management**: the SDK registers topics for exclusive access, as well as enables users to view and modify various configurations of registered topics.
- **Trigger Management**: the SDK facilitates users to create, view, update, and delete Octopus triggers, enabling real-time event processing without configuring traditional consumers.

Expected Reproduction Time (in Minutes)

The notebook walk-through will take approximately fifteen minutes, covering SDK installation, execution of notebook cells to engage with the hosted Octopus Web Service, and waiting times for operations such as trigger creation and deletion.

Artifact Setup (incl. Inputs)

The notebook in the artifact does not require a particular environment or specific hardware. The first code cell installs the Octopus SDK and all required dependencies from PyPI. This notebook has been tested with Python 3.9.6 and Python 3.10.14. It is recommended that the notebook be executed in a Python 3.8 or higher virtual environment.

Artifact Execution

Following the SDK installation, the second code cell performs user authentication via Globus Auth. The platform supports a wide range of identities from various organizations, including academic institutions, GitHub, Google, and ORCID.

The first section of the notebook demonstrates explicitly acquiring a connection credential for the event fabric cluster. After acquiring the credential and storing it in local token storage, the code produces a message to the event fabric and consumes another from a public topic by calling `block_until_ready()`.

The second section covers topic management APIs: acquiring and releasing topic access, adjusting topic settings, and using the SDK producer and consumer on a registered topic.

The third section introduces APIs for trigger creation, listing, updating, and deletion. Following the code cells, the user can create a basic trigger that records incoming events, create and remove the filter described in Listing 1 of the paper, and tear the trigger down with associated AWS resources. To verify the trigger has been invoked, the code cells provide execution logs and trigger prints fetched from AWS CloudWatch.

Artifact Analysis (incl. Outputs)

For all SDK methods, we list the formats of the expected return in the comments before the code. For methods calling Octopus Web Service, we expect the responses to show `"status": "success"` with requested information. Some special cases include: 1) In the first section, we expect the assertion to return without raising an exception. 2) In the second section, we expect the SDK consumer to consume messages produced earlier, which may arrive out of order if the topic has been configured with more than one partition by the user.

B. Computational Artifact A_2

Relation To Contributions

This artifact details our benchmarking methodology, including the code and setup for the benchmarking producer and consumer, Octopus trigger, and Parsl. It also includes raw benchmarking datasets and the scripts used for data analysis and visualization. The documentation supports the findings in the evaluation and application sections and illustrates how the figures and tables were generated.

Expected Results

We expect the provided Python notebook to reproduce Figures 3-5, 7, and 8, as well as Table 3, using the raw data in the artifact. In the description below, we outline the complete benchmarking process and the steps for data collection beyond what has been discussed in the paper.

Expected Reproduction Time (in Minutes)

We estimate that the time to reproduce these tables and figures using our provided notebook is fifteen minutes.

Artifact Setup (incl. Inputs)

Producer and Consumer Benchmarking: We modify Kafka 3.5.1's benchmarking producer and consumer to support duration-based benchmarking; That is, the producers and consumer run until the specified time duration (e.g., two minutes) is passed. At each client instance (i.e., on AWS and Chameleon Cloud), we clone and check out Kafka 3.5.1 from the official GitHub repository, replace the source code of the benchmarking clients with our code, and recompile the Kafka library.

On the benchmarking operator side, we clone the GitHub repository that the benchmarking shell scripts belong to and specify the IPs of the client machines as well as SSH keys to connect in `remote-configs.sh`. We also specify the number of repeats and other parameters (number of producers, event sizes, etc.) in `run-remote.sh`. When we execute this script, it contacts all remote clients to spawn the desired number of producers or consumers. When a round of benchmarking is complete, the script automatically collects logs from client instances. The collected logs are in `group1` to `group7` folders in `bmk-data`, categorized by the cluster specifications and whether clients are remote (on Chameleon Cloud instances) or local (on AWS EC2 instances).

Octopus Trigger Benchmarking: For trigger-related benchmarking, we invoke our trigger creation API to set triggers with specified function code and invocation parameters. For the scaling test, the function sleeps for 30 seconds upon receiving an event from a topic with 128 partitions. For the Scientific Data Automation app, the function prints the received event from a topic with 8 partitions and invokes the Globus Transfer service to initiate the data transfer. The function waits until the transfer task is complete. The invocation batch sizes of both functions are capped to one, but the trigger for Scientific Data Automation receives file creation events batched previously by the local aggregator. We downloaded the execution log streams from AWS CloudWatch; they are in `trigger8topics` and `trigger128topics` folders in `bmk-data`.

Parsl Benchmarking: We use Texas A&M's FASTER cluster, a cluster with Intel(R) Xeon(R) Platinum 8352Y CPUs @ 2.20GHz. In this experiment, we used 4 computation nodes, all using the Rocky Linux 8 Linux distribution.

The Parsl library we use here is a forked version, which can be found in the `parsl_resilient` folder of the artifact. Use the following command to install this library and the dependencies.

- `cd parsl_resilient`
- `pip install -e .`
- `pip install 'parsl[monitoring]'`

The test script `run_one.py` in the artifact takes in arguments such as the number of workers, the number of tasks, and the monitor mode to generate Parsl configurations and run those tasks. When all of the tasks finish and return, it will record the task execution time and insert the corresponding information into the database. `strong.sh` script invokes

`run_one.py` at different configurations and records the total makespan of each Python program.

To plot Figure 8 on the paper, we need to calculate the number of events and the overhead for each trail. `cal_entry.py` reads `run_id` from the Parsl database and uses them to calculate the number of events. `cal_overhead.py` subtracts the task execution time from makespan to get the overhead, which is stored in `FASTERdata.db` in `bmk-data`.

Python Notebook: The Python notebook for plotting figures and generating Table 3 requires no special hardware. The first code cell installs necessary PyPI packages for plotting. We have tested the notebook with Python versions 3.9.6 and 3.10.14. Additionally, a `requirements.txt` is provided for Python 3.9.6; however, the first code cell should install all required dependencies without a problem.

Artifact Execution

In this section, we explain how the provided notebook uses the collected data for plotting and generating Table 3. The notebook is structured into seven sections. The first section installs dependencies and loads the benchmarking producer and consumer logs to variables for plotting later, and executing through each of the following sections produces a figure or a table on the paper. Notebook sections are organized in the order in which these elements appear on the papers. Specifically, Sections 2 through 4 plot Figures 3 to 5. Section 5 generates Table 3, showing throughput and latency metrics. Finally, Sections 6 and 7 produce Figures 7 and 8.

Artifact Analysis (incl. Outputs)

We expect the figures generated throughout the notebook to match closely with those in our paper, specifically Table 3, Figures 3-5, 7, and 8. The other figures included in the paper do not show experiment results.