## DISCLAIMER

**This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Reference herein to any social initiative (including but not limited to Diversity, Equity, and Inclusion (DEI); Community Benefits Plans (CBP); Justice 40; etc.) is made by the Author independent of any current requirement by the United States Government and does not constitute or imply endorsement, recommendation, or support by the United States Government or any agency thereof.**

ORNL/TM-2025/4190

# Deployment and Evaluation of SciStream on OLCF's Advanced Computing Ecosystem (ACE)


Anjus George

**December 2025**

**Approved for public release.**
**Distribution is unlimited.**

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

**ORNL/TM-2025/4190**

National Center for Computational Sciences

# DEPLOYMENT AND EVALUATION OF SCISTREAM ON OLCF'S ADVANCED COMPUTING ECOSYSTEM (ACE)

Anjus George

December 2025

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

## CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ACE | Advanced Computing Ecosystem |
| AMQP | Advanced Message Queuing Protocol |
| ANL | Argonne National Laboratory |
| APS | Advanced Photon Source |
| CN | Common Name |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| DOE | U.S. Department of Energy |
| DSNs | Data Streaming Nodes |
| DTNs | Data Transfer Nodes |
| ESnet | Energy Sciences Network |
| GNs | Gateway nodes |
| GRETA | Gamma-Ray Energy Tracking Array |
| HPC | High-performance computing |
| IRI | Integrated Research Infrastructure |
| LAN | Local area network |
| LCLS | Linac Coherent Light Source |
| NAT | Network Address Translation |
| ODPs | On-demand proxies |
| OLCF | Oak Ridge Leadership Computing Facility |
| RTT | Round-trip time |
| S2CS | SciStream Control Server |
| S2DS | SciStream Data Server |
| S2UC | SciStream User Client |
| SAN | Subject Alternative Name |
| SNS | Spallation Neutron Source |
| TLS | Transport Layer Security |
| UIDs | Unique identifiers |
| WAN | Wide-area network |

## ABSTRACT

The growing demand for real-time analysis, experimental steering, and decision-making in scientific workflows has created a need for tightly coupled integrations between experimental facilities and high-performance computing (HPC) systems. The Department of Energy's Integrated Research Infrastructure (IRI) initiative highlights data streaming as a key capability for enabling memory-to-memory data transfers, bypassing the limitations of traditional store-and-forward models. SciStream is a toolkit developed by researchers at Argonne National Laboratory (ANL) to support such streaming by addressing cross-domain security, delegated authentication, and application transparency. We deployed and evaluated SciStream on the Oak Ridge Leadership Computing Facility's (OLCF) Advanced Computing Ecosystem (ACE) infrastructure, leveraging the Olivine OpenShift cluster and its high-bandwidth Data Streaming Nodes (DSNs) as gateway nodes. Our evaluation included synthetic streaming workloads derived from IRI science workflows, a streaming simulator, and integration with RabbitMQ to handle low-level messaging. This report documents the deployment process, performance evaluation, and challenges encountered, along with opportunities for future improvements.

## 1. INTRODUCTION

The scientific workflow landscape is evolving rapidly, with modern workflows increasingly requiring near real-time data analysis, experimental steering, and informed decision-making during experiment execution. This shift is driving the development of tightly coupled workflows that integrate scientific instruments at experimental facilities with High-performance computing (HPC) systems, enabling on-the-fly analysis of experimental data and timely feedback to guide experimental control. Examples include the Linac Coherent Light Source (LCLS) workflow at SLAC National Accelerator Laboratory, which streams diffraction frames from the LCLS light source over Energy Sciences Network (ESnet) [7] directly to HPC systems at Oak Ridge Leadership Computing Facility (OLCF) [35]; the AI-enabled workflow at Argonne's Advanced Photon Source (APS), which uses HPC nodes to retrain models on the fly for real-time experimental steering [6]; and the edge-to-exascale workflow using Frontier at OLCF, where a Temporal Fusion Transformer (TFT) at the Spallation Neutron Source (SNS) predicts 3D scattering patterns and provides feedback to adjust beam settings [43].

The U.S. Department of Energy (DOE) Integrated Research Infrastructure (IRI) initiative aims to accelerate the time to insight from such tightly coupled experimental-HPC workflows [26]. In its 2023 report [31], the IRI task force identified several templates for interacting with HPC resources. Among these, data streaming offers powerful emerging capabilities such as near real-time data analysis, experimental steering, and informed decision-making during experiment execution. Unlike traditional store-and-forward models, which involve writing data to disk, transferring it over wide-area filesystems, and then reading it on a supercomputer, data streaming enables direct memory transfers. With direct memory streaming, bytes are moved straight from an edge node's DRAM into an HPC job's address space, bypassing intermediate storage layers.

SciStream [12] is one such tool that tackles the infrastructural challenges necessary to enable these memory-to-memory data transfers between instruments and HPC, even when neither system has direct network connectivity. It treats each experimental or HPC facility as an independent security domain and allows each to expose streaming resources through a federated interface. SciStream addresses three primary challenges: handling data transfers across security domains between data producers (e.g., scientific instruments) and consumers (e.g., HPC systems); supporting delegated authentication and authorization within broader scientific workflows; and decoupling sophisticated identity and access management from applications, minimizing changes and reusing existing security architectures.

An initial discussion with the SciStream team at Argonne National Laboratory (ANL) led to a collaborative effort to deploy it on the OLCF's Advanced Computing Ecosystem (ACE) infrastructure [36]. The ACE infrastructure at OLCF serves as a platform to support various foundational technologies, computing capabilities and cutting-edge science. SciStream's architecture relies on Gateway nodes (GNs) that act as intermediaries, connecting the internal instrument/HPC network with the external Wide-area network (WAN). The Olivine OpenShift [27] cluster within ACE includes high-bandwidth Data Streaming Nodes (DSNs) [8] with both internal and external 100 Gbps connectivity, making them ideally suited to serve as SciStream GNs.

The first phase of the project focused on preparing SciStream components for deployment on the OpenShift environment of the Olivine cluster. After making the components OpenShift-compatible, we tested them extensively and deployed them on the Olivine DSNs. In the next phase, we evaluated the functionality of the different components and the options that SciStream provides. To assess performance and scalability under conditions resembling real streaming workloads, we derived synthetic workloads from the streaming characteristics of two IRI science workflows—Gamma-Ray Energy Tracking Array (GRETA)/Deleria

[13] and LCLS [40]. Producer and consumer applications were deployed on OLCF's Andes [32] compute cluster to test the scalability of streaming through SciStream. In addition, we developed a streaming simulator that emulates experiments in which these synthetic workloads are streamed from producers to consumers. We also deployed a streaming service, such as RabbitMQ [10], to handle low-level messaging aspects during the SciStream experiments.

This technical report documents our efforts beginning with the deployment, testing, and evaluation of SciStream on ACE. §2 introduces SciStream, its design goals, and architectural components. §3 describes the containerized deployment of SciStream on a local system to gain familiarity with its commands and setup. §4 provides details of the ACE platform and deployment hardware. §5 outlines the step-by-step deployment process for two setups: one with producer applications deployed locally communicating with consumer applications on the Olivine cluster, and another with both producer and consumer applications deployed on the Andes cluster and communicating through a RabbitMQ streaming service. §6 presents our evaluation of SciStream using the second setup through simulated streaming experiments. §7 concludes the report, discusses the challenges faced during deployment and testing, along with future work.

## 2. ARCHITECTURE AND COMPONENTS OF SCISTREAM

SciStream [12, 16] is a middlebox-based architecture and toolkit designed to enable secure, efficient, and transparent memory-to-memory data streaming between scientific instruments and remote HPC facilities. Unlike traditional file-based data staging that relies on intermediate storage systems, SciStream directly bridges producer and consumer applications across multiple security domains, overcoming restrictions due to firewalls, Network Address Translation (NAT), and limited external connectivity of instruments and compute nodes.

SciStream was developed by researchers from ANL with four main design goals in mind. The first goal is *third-party streaming*. In scientific workflows, it is often desirable for users or workflow engines (e.g., Swift/T [42], Pegasus [15], Galaxy [25]) to initiate, manage, and monitor data transfers without requiring direct intervention from the producer or consumer applications themselves. SciStream enables this model by separating the control and data channels. The control channel is responsible for authentication, authorization, and negotiation, while the data channel carries the actual data streams. This design ensures flexibility, transparency, and fault isolation, making it easier to integrate SciStream into large-scale workflows.

The second goal is *secure streaming*. Because instruments and HPC systems typically reside in distinct administrative and security domains, end-to-end security is critical. SciStream enforces authentication and authorization on both control and data connections. It leverages federated identity systems like InCommon [41] and Globus Auth [11] to manage user access seamlessly across facilities. For the control plane, SciStream uses shared-key authentication, while for the data plane it relies on source-based authentication. Encryption can also be enabled on demand to further secure sensitive communications across WANs.

The third goal is *general and transparent streaming*. SciStream is designed to be application-agnostic, requiring minimal changes to existing producer or consumer code. This is achieved by using transport-layer (Layer-4) proxies that are transparent to applications and independent of the specific streaming library in use–whether ZeroMQ [5], RabbitMQ, Kafka [20], or others. This approach avoids the complexity and scalability limitations of application-layer (Layer-7) proxies, which must support each protocol individually, and the deployment challenges of network-layer (Layer-3) NAT or tunnels, which can be difficult to scale and may introduce additional overhead.

Finally, SciStream supports both *provisioned and best-effort resources*. In provisioned scenarios, resources such as bandwidth and compute capacity are reserved in advance to guarantee performance. In contrast, best-effort scenarios arise in environments where HPC jobs may be queued or network traffic is shared with other workloads. SciStream's control protocol is flexible enough to handle both cases by allowing asynchronous setup. Producers, consumers, and controllers can connect independently when they become available, enabling both time-sensitive guaranteed workflows and opportunistic data streams to coexist effectively.

As shown in Figure 1, SciStream's architecture leverages the Science DMZ [14] to deploy On-demand proxies (ODPs) on specialized GNs, which act as bridges between internal instrument or HPC networks and the external WAN. These proxies are implemented as Layer-4 TCP proxies with reconfigurable circular buffers. While dedicated GNs provide superior performance and easier management by avoiding contention from file-based transfers, re-purposing existing Data Transfer Nodes (DTNs) can be a cost-effective alternative, particularly for smaller institutions. SciStream is designed to support both deployment models, making it adaptable across a wide range of scientific facilities.

SciStream's architecture is realized through three core components: *SciStream User Client (*S2UC*)*, *SciStream Control Server (*S2CS*)*, and *SciStream Data Server (*S2DS*)* (see Figure 2). The S2UC is the interface

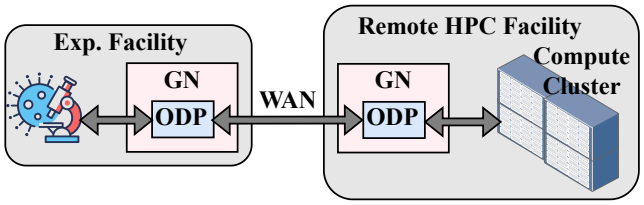


**Figure 1. SciStream's architecture for memory to memory data streaming. On-demand proxies (ODP) are deployed on specialized Gateway nodes (GNs) in both facilities.**

through which users or workflow systems request, manage, and monitor streaming sessions. It handles user authentication, resource negotiation, and connection orchestration, while also providing monitoring information such as throughput and channel status. The S2CS operates on GNs within the facilities. Its role is to authenticate users and applications, manage resources, and launch S2DS processes, while also mapping producer-consumer associations to ensure resources are allocated correctly. The S2DS runs on GNs and functions as the proxy between the local network of instruments or HPC interconnects and WAN. It uses proxy certificates to authenticate external WAN connections and source-based authentication for internal Local area network (LAN) or HPC connections. Its primary purpose is to forward data streams while maintaining transparency to producer and consumer applications.

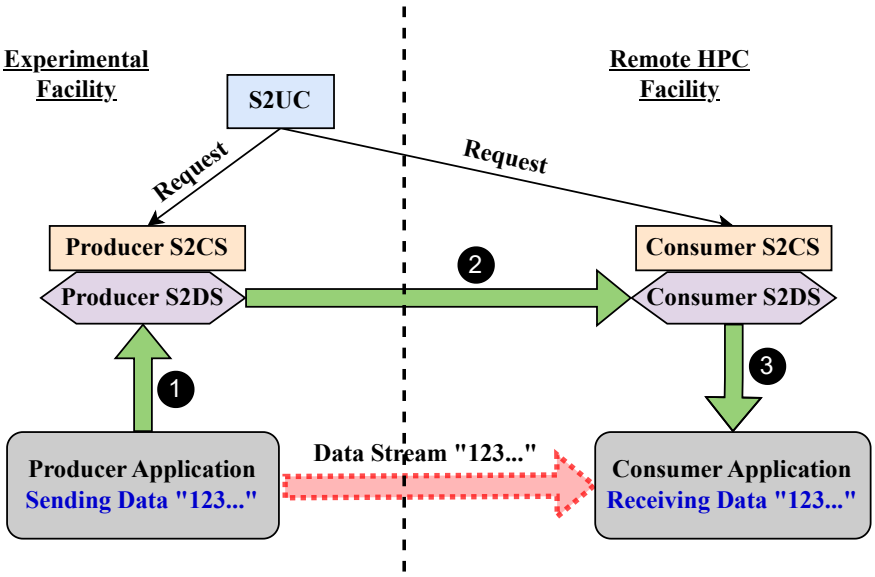


**Figure 2. SciStream's components and data flow paths.**

SciStream introduces two major protocols to manage its operation: a negotiation protocol and a control protocol. The negotiation protocol ensures that, before streaming begins, the producer and consumer agree on the required bandwidth and number of channels through their respective S2CS. If necessary, additional channels can be allocated to meet bandwidth demands, ensuring that both facilities can support the requested stream and that resources are distributed in a balanced manner. Once negotiation is complete, the control protocol orchestrates the end-to-end setup. The S2UC collects user requests, authenticates with

facilities, and assigns a unique identifier. Each S2CS on GNs then instantiates new S2DS to act as proxies bridging local and remote networks. Ports are allocated, connection maps are constructed, and both producers and consumers are informed of their designated endpoints. Once mappings are distributed, the proxies establish connections, and the data path is activated. The data then flow through three transparent hops: producer application → producer proxy (producer S2DS) → consumer proxy (consumer S2DS) → consumer application, as indicated by ❶ , ❷ , and ❸ in Figure 2. From the applications' perspective, these paths are fully transparent, and the data stream appears to flow directly from the producer application to the consumer application (indicated by the red arrow). The control protocol also provides teardown mechanisms to release resources once a request is complete.

## 3. LOCAL SCISTREAM SETUP USING DOCKER

This section outlines the steps required to set up SciStream using Docker [30] and establish communication between a simple producer and consumer application. This process is intended to familiarize with SciStream deployment options and commands. Figure 3 illustrates this setup and the component interactions. The only prerequisite is to have Docker installed locally.

For this deployment, we used the latest SciStream toolkit (version 1.2.1) [18, 17], whose Docker image is available on Docker Hub [29]. The image is first pulled to the local environment, then tagged with the address of the `registry.apps.olivine.ccs.ornl.gov` registry under a namespace (e.g., `stf008`). It is then pushed to the registry so it can be reused. The remaining process is divided into five steps.

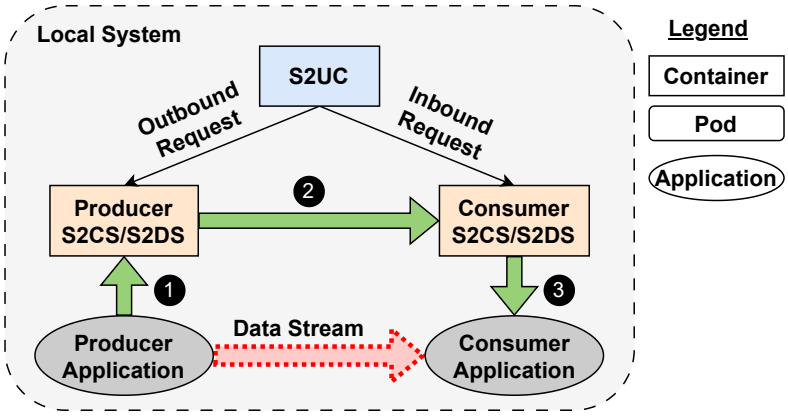


**Figure 3. SciStream setup on a local system using Docker containers.**

### 3.1 START CONSUMER APPLICATION

In this setup, we use a simple Python TCP server as the consumer application. It listens on all interfaces at port `8080`. Start the consumer application with:

```
python3 consumer-app.py
```

### 3.2 SET UP AND START PRODUCER S2CS

In this step, the producer-side S2CS is set up and started.

1. Run the SciStream Docker image in an interactive terminal with Bash as the entrypoint. It also maps port `5000` and the range `5100-5110` from the container to the host, allowing services inside the container to be accessed on those ports. Port `5000` is used by S2CS for control-plane requests, while all other ports are used by S2DS depending on the number of connections specified in the S2UC requests.

   ```
   docker run -it -p 5000:5000 -p 5100-5110:5100-5110  --entrypoint
   /bin/bash registry.apps.olivine.ccs.ornl.gov/stf008/scistream:1.2.1
   ```

2. Find the container's routable interface for communication between the container, host, and external network. For most containers it is the primary Ethernet interface (e.g., `eth0`), connected to Docker's default bridge network. Here `172.17.0.2` is the IP assigned to the container by Docker's internal Dynamic Host Configuration Protocol (DHCP) for that network. Next generate a self-signed X.509 certificate (`prod-server.crt`) and private key (`prod-server.key`) using a 2048-bit RSA key. Set the certificate's Common Name (CN) and Subject Alternative Name (SAN) to the IP address `172.17.0.2` of the container, so it can be used to authenticate a service running at that address.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
prod-server.key -out prod-server.crt -subj "/CN=172.17.0.2" -addext
"subjectAltName=IP:172.17.0.2"
```

3. Next, start producer S2CS using the self-signed certificate and key that were generated earlier. SciStream supports multiple data server implementations, including HAProxy, Nginx, Stunnel, StunnelSubprocess, and HaproxySubprocess. In this setup, we use the StunnelSubprocess type of data server and specify the container IP as `listener_ip` so that the server can listen on it.

```
s2cs --server_crt="prod-server.crt" --server_key="prod-server.key"
--type="StunnelSubprocess" --verbose --listener_ip=172.17.0.2
```

### 3.3 SET UP AND START CONSUMER S2CS

This step is similar to the previous one, as we start another SciStream control server, this time the consumer S2CS, configured in the same way as the producer S2CS. The main differences are that we use a different host port range (since ports `5000` and `5100-5110` are already mapped to the producer S2CS container), the container has a different IP address, and we generate a separate certificate and key for the consumer. The corresponding commands are shown below.

```
docker run -it -p 6000:5000 -p 6100-6110:5100-5110  --entrypoint /bin/bash
registry.apps.olivine.ccs.ornl.gov/stf008/scistream:1.2.1

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout cons-server.key
-out cons-server.crt -subj "/CN=172.17.0.3" -addext
"subjectAltName=IP:172.17.0.3"

s2cs --server_crt="cons-server.crt" --server_key="cons-server.key"
--type="StunnelSubprocess" --verbose --listener_ip=172.17.0.3
```

### 3.4 START S2UC AND SEND REQUESTS

Next, we start a container for the SciStream S2UC and send requests to it, which are then relayed to the producer and consumer S2CS. Upon receiving a request, the S2UC generates a unique ID and passes it to both the producer and consumer S2CS. Once the S2CSs accept the request, they create S2DS proxies that stream data between each other. The request sent to the consumer S2CS is called an *inbound request*, while the request sent to the producer S2CS is called an *outbound request*. Note that the certificates generated by the producer and consumer S2CS in the previous steps must be shared with the S2UC so it can authenticate the connections.

1. Run the SciStream Docker image in an interactive terminal. Mount the local `./certs` directory, containing the producer and consumer S2CS certificates copied from their respective containers, into the S2UC container at `/certs`, making the certificate files available inside the container.

```
docker run -it -v ./certs:/certs --entrypoint /bin/bash
registry.apps.olivine.ccs.ornl.gov/stf008/scistream:1.2.1
```

2. Send the inbound request to the consumer S2CS by specifying the server certificates and the container's IP address (`172.17.0.3`). The `remote_ip` indicates the next hop for the data from the consumer S2DS, which in this case is the consumer application (see Figure 3) listening on `0.0.0.0:8080` (see §3.1) of the host machine. Because the request originates inside a container, `host.docker.internal` is used as the `remote_ip`, since it is a special Domain Name System (DNS) name provided by Docker that resolves to the host machine's IP address. The `receiver_ports` parameter is set to `8080`, and the number of connections is one.

   ```
   s2uc inbound-request --server_cert="/certs/cons-server.crt" --remote_ip
   host.docker.internal --s2cs 172.17.0.3:5000 --receiver_ports 8080
   --num_conn 1
   ```

   As a result of the inbound request, the consumer S2CS creates a consumer S2DS proxy (opens a port, `PROXY_C`) and assigns it a unique ID (`UID`).

3. Next, send the outbound request to the producer S2CS by specifying the server certificates and the container's IP address (`172.17.0.2`). The `remote_ip` is the consumer S2CS IP address (`172.17.0.3`), which serves as the next hop (see Figure 3). The `receiver_ports` parameter is set to the port opened (`PROXY_C`) by the consumer S2CS as a result of the inbound request, and `num_conn` is set to one. The command also includes the `UID` generated by the consumer S2CS, along with its IP address and proxy (`PROXY_C`).

   ```
   s2uc outbound-request --server_cert="/certs/prod-server.crt" --remote_ip
   172.17.0.3 --s2cs 172.17.0.2:5000 --receiver_ports PROXY_C --num_conn 1
   UID 172.17.0.3:PROXY_C
   ```

   As a result of the outbound request, the producer S2CS creates an S2DS proxy (`PROXY_P`).

### 3.5 STREAM DATA

The final step is to send data from the producer application to the consumer application. Since the consumer is a simple Python TCP server, the producer can be as simple as a Netcat [28] client to send messages to it.

```
echo "123" | nc 127.0.0.1 PROXY_P
```

The reason the localhost IP `127.0.0.1` is used to send data to the producer S2DS instead of the producer S2CS container IP (`172.17.0.2`) is that the host machine cannot directly access a container's internal IP address. While containers on the same Docker network can communicate with each other using their container IPs, the host must rely on port mappings. In this case, port `PROXY_P` inside the container is mapped to the same port on the host machine (see §3.2), so using `127.0.0.1` allows the host to reach the container through the mapped port.

## 4. DEPLOYMENT PLATFORM AND SPECIFICATIONS

The Advanced Computing Ecosystem (ACE) is a strategic initiative within the OLCF that supports the development of cutting-edge technologies to advance scientific research and innovation at OLCF and across the DOE [36]. In addition to computing environments that span moderate (e.g., Frontier [2] and Andes [32]) and open (e.g., Odo [33]) enclaves, ACE also provides a testbed offering a centralized, sandboxed environment that incorporates emerging compute and storage architectures. In this section, we describe the specific components used for SciStream deployment and evaluation on the ACE testbed.

### 4.1 OLIVINE OPENSHIFT CLUSTER

Within the ACE infrastructure, the Olivine OpenShift cluster [34] provides a suitable platform for streaming technology deployments, as it includes DSNs, dedicated gateway hosts that bridge the public WAN and internal OLCF networks via high-speed network adapters. For this reason, we selected the DSNs to host SciStream's control and data servers. Purpose-built for streaming, each DSN is equipped with two 32-core 2.70 GHz AMD EPYC 9334 processors and 512 GiB of RAM. While each node supports 100 Gbps connectivity to both internal and external networks, current usage is limited to 1 Gbps due to ongoing efforts to fully configure high-speed interfaces within the OpenShift environment. Besides the DSNs, the Olivine cluster also includes 13 standard nodes without high-speed connectivity. Among these, some are configured with 64 cores and 256 GB of RAM, while others have 128 cores and 512 GB of RAM; all of them natively support only 10 Gbps networking.

### 4.2 ANDES COMPUTE CLUSTER

To deploy producer and consumer applications communicating through SciStream, we utilize OLCF's Andes [32] cluster. Each Andes node is equipped with two 16-core 3.0 GHz AMD EPYC 7302 processors and 256 GiB of RAM. Andes and the DSNs in Olivine cluster are connected via a 1 Gbps Ethernet network.

## 5. DEPLOYMENT ON ACE

To deploy SciStream on ACE, we follow the same five steps described in §3, but adapt them for the OpenShift-based Olivine cluster in ACE. The producer S2CS and consumer S2CS are deployed as OpenShift Pods. We describe two types of setups: in the first, the producer runs on a local machine and communicates with a consumer application running (in a Pod) on the Olivine cluster; in the second, both producer and consumer applications run on Andes compute nodes. In the second setup, we also deploy and use an off-the-shelf streaming service as a front end to SciStream, enabling the producer and consumer applications to exchange data through this service.

### 5.1 SETUP 1: LOCAL PRODUCER AND OLIVINE CONSUMER

Figure 4 illustrates setup 1 where a local producer streams data to an Olivine consumer through SciStream.

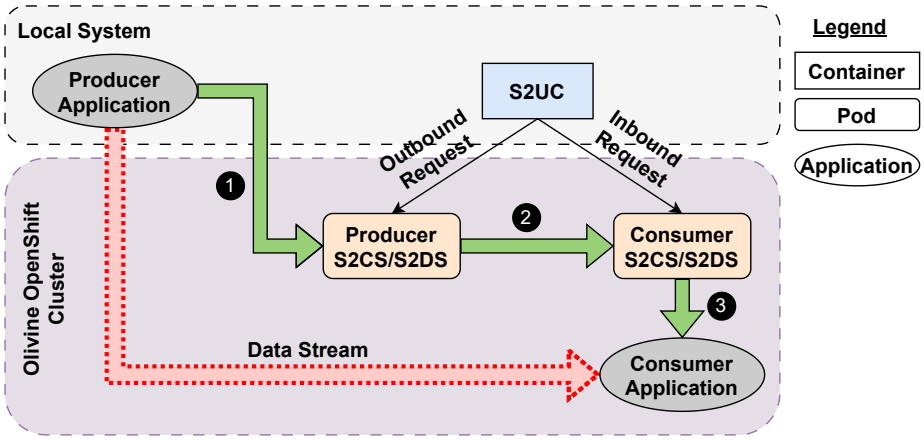


**Figure 4. Setup 1: Local producer and Olivine consumer interactions with SciStream components deployed on the Olivine OpenShift cluster.**

#### 5.1.1 Start Consumer Application

1. In this setup, we first define a Pod for the consumer application in a YAML file named `consumer-pod.yaml`. This Pod runs the same Python TCP server consumer application described in §3.1. The Pod definition creates a Pod named `consumer-pod` with a single container based on the SciStream Docker image. The container mounts a scratch volume at `/scratch` inside the container. Alternatively, any Linux-based Docker image with Python installed could be used to run the simple Python TCP server instead of the SciStream image. We then create the Pod from the YAML definition:

   ```
   oc create -f consumer-pod.yaml
   ```

   This deploys the Pod on one of the standard Olivine nodes. Let us refer to the IP address of this Pod as `CONS_APP_POD_IP`.

2. Next, we copy the consumer application `consumer-app.py` from the local machine into the running Pod, placing it under `/scratch/consumer-app.py`:

   ```
   oc cp ./consumer-app.py consumer-pod:/scratch/consumer-app.py
   ```

3. Finally, we start the consumer application inside the Pod, which begins listening on port `8080`.

### 5.1.2 Set up and Start Producer S2CS

Similar to the consumer application, the producer S2CS is also defined as a Pod to run in the OpenShift environment. The Pod's YAML definition (`producer-s2cs.yaml`) creates a SciStream producer S2CS Pod and runs a container from the SciStream image specifically on one of the Olivine DSNs. In §3.2, we ran a container with the required host-to-container port mappings using Docker's CLI; here, the Pod definition instead exposes the same mappings (port `5000` for control and ports `5100-5110` for the data plane) through Kubernetes. The remaining steps to launch the producer S2CS, such as certificate generation and starting the S2CS instance, are provided as container arguments. When the container starts, it generates a self-signed certificate bound to the Pod's IP and launches the S2CS process listening on that IP. In addition, the YAML defines an OpenShift Service resource [4] called `prod-s2cs-service`. A Service provides a stable network endpoint to access the Pod, even though Pod IPs are dynamic and not directly reachable from outside the cluster. Deploy the Pod and start the producer S2CS with:

```
oc create -f producer-s2cs.yaml
```

Let us call the IP of this Pod, where the producer S2CS is deployed, `PROD_S2CS_POD_IP`.

### 5.1.3 Set up and Start Consumer S2CS

Similar to the producer S2CS in the previous step, we create a Pod definition (`consumer-s2cs.yaml`) for the consumer S2CS, then deploy and launch it.

```
oc create -f consumer-s2cs.yaml
```

Let us call the IP of this Pod, where the consumer S2CS is deployed, `CONS_S2CS_POD_IP`, and refer to the Service defined for the Pod as `cons-s2cs-service`.

### 5.1.4 Start S2UC and Send Requests

In this step, we start a container locally for SciStream S2UC in the same way described in §3.4. As before, the certificates generated by the producer and consumer S2CS in the previous steps must be shared with the S2UC so it can authenticate the connections. The following command can be used to copy the certificates from their Pods to the local machine:

```
oc cp pod-name:/certs/server-cert.crt ./server-cert.crt
```

1. Start the S2UC container locally. The following command runs the S2UC container with the directory `./s2uc-mount` mounted to `/certs`, where the producer and consumer S2CS certificates have been copied:

   ```
   docker run -it -v ./s2uc-mount:/certs --entrypoint /bin/bash
   registry.apps.olivine.ccs.ornl.gov/stf008/scistream:1.2.1
   ```

2. The next step is to send the inbound request to the consumer S2CS. Unlike the fully local setup of SciStream described in §3, where both S2UC and the S2CS processes ran in local Docker containers, here the S2UC container is local while the S2CS processes run as Pods in the OpenShift cluster. Since Pod IPs are internal to the cluster and cannot be accessed directly from outside, we use port forwarding. Port forwarding creates a secure tunnel between the local environment and the service inside the cluster, allowing interaction as if the service were running locally. In this case, we forward port `5000` on the local machine to port `5000` of the `cons-s2cs-service` (created in §5.1.3) inside the cluster.

   ```
   oc port-forward -n stf008 svc/cons-s2cs-service 5000
   ```

This allows interaction with the consumer S2CS as if it were running locally on `127.0.0.1:5000`. The inbound request can then be sent with:

```
s2uc inbound-request --server_cert="certs/cons-server.crt"
--remote_ip CONS_APP_POD_IP --s2cs host.docker.internal:5000
--receiver_ports 8080 --num_conn 1
```

Here, `CONS_APP_POD_IP` is the Pod IP of the consumer application listening on port `8080`, which is the next hop from the consumer S2CS Pod. The consumer S2CS itself is specified as `host.docker.internal` instead of `CONS_S2CS_POD_IP`, since Pod IPs are not reachable outside the cluster. The special DNS name `host.docker.internal` resolves to the host machine's `127.0.0.1`, and with port forwarding enabled, traffic sent to `127.0.0.1:5000` is routed through the `cons-s2cs-service` to the consumer S2CS Pod. As a result of this inbound request, the consumer S2CS creates a consumer S2DS proxy (`PROXY_C`) and assigns it a unique ID (`UID`).

3. Send the outbound request to the producer S2CS. In the same way, we first forward local port `5000` to the `prod-s2cs-service`:

```
oc port-forward -n stf008 svc/prod-s2cs-service 5000

s2uc outbound-request --server_cert="certs/prod-server.crt"
--remote_ip CONS_S2CS_POD_IP --s2cs host.docker.internal:5000
--receiver_ports PROXY_C --num_conn 1 UID CONS_S2CS_POD_IP:PROXY_C
```

Here, `CONS_S2CS_POD_IP` is the next hop from the producer S2CS Pod. As with the inbound request, `host.docker.internal` is used along with port forwarding to communicate with the producer S2CS. As a result of this outbound request, the producer S2CS creates an S2DS proxy (`PROXY_P`).

#### 5.1.5 Stream Data

To send a data stream from the local producer, we use the same Netcat client to transmit data to the Python TCP server consumer application. Data can be sent either from outside a container (on the local machine) or from inside a container. Before sending data, port forwarding to the `prod-s2cs-service` through port `PROXY_P` must be enabled:

```
oc port-forward -n stf008 svc/prod-s2cs-service PROXY_P
```

- From outside a container, send data to the consumer application with:

```
echo "123" | nc 127.0.0.1 PROXY_P
```

- From inside a container, send data using:

```
echo "123" | nc host.docker.internal PROXY_P
```

### 5.2 SETUP 2: ANDES PRODUCERS AND CONSUMERS

In this setup, we use nodes in OLCF's Andes compute cluster to deploy the producer and consumer applications. In a real experimental data streaming scenario, producers typically run on clusters or machines located either in a different facility or in a separate cluster within the same facility. Such streaming scenarios also involve low-level messaging aspects such as connection management, buffering, and flow control between producers and consumers. To address this, we deploy a dedicated backend streaming service, such as RabbitMQ, as a front end to SciStream. RabbitMQ is a messaging broker that implements the Advanced Message Queuing Protocol (AMQP) [9] as its wire-level communication protocol. A RabbitMQ server

(broker) enables clients to send, receive, or temporarily store messages using queues (i.e., ordered collections of messages that are held until consumed). Figure 5 illustrates this setup. Unlike the previous setups, the data flow path (shown in green) includes an additional fourth hop: from the RabbitMQ server to the receiving consumer application. Furthermore, as seen in the previous setup (§5.1), sending requests or data from outside the OpenShift cluster requires enabling port forwarding for each opened port or proxy. To simplify this, we instead use NodePorts, a Kubernetes Service type [4] that exposes a Pod's port on a fixed port of every cluster node, allowing external clients to connect directly to `<NodeIP>:<NodePort>` without requiring port forwarding.

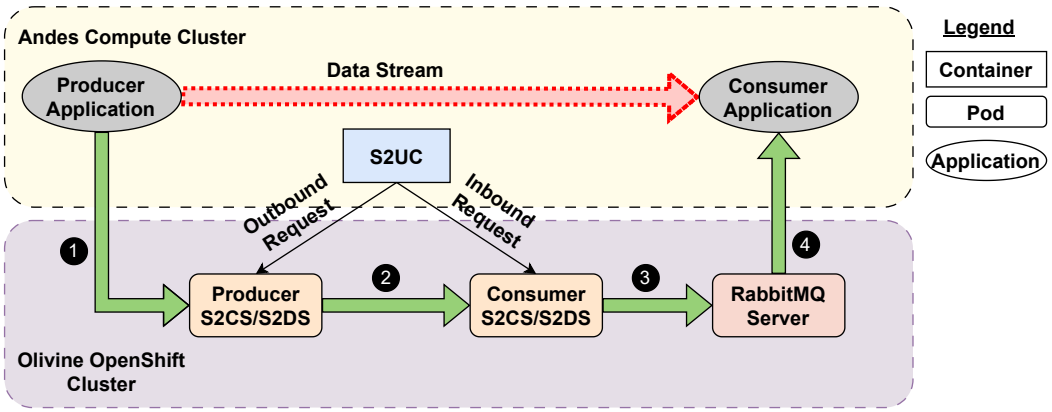


**Figure 5. Setup 2: Producer and consumer applications on Andes cluster communicating with RabbitMQ server through SciStream deployed on the Olivine OpenShift cluster.**

### 5.2.1 Deploy Streaming Service

To deploy a three-server RabbitMQ streaming service, we used the Bitnami RabbitMQ Helm Chart (version v3.18.3) [21]. This chart provisions a RabbitMQ cluster with servers running as separate Pods on OpenShift, each configured with specific security, performance, and networking settings. Transport Layer Security (TLS) is enabled using auto-generated certificates, and pod anti-affinity rules ensure that the three RabbitMQ server Pods are scheduled on separate Olivine DSNs. Each replica requests 12 CPUs and 32 GiB of memory and is allocated 15 GiB of persistent storage. The service is exposed via NodePort, with ports `30672` (AMQP) and `30671` (AMQPS) made available on the cluster nodes. These NodePorts allow external access from both producers and consumers. Let us refer to the node/host IP of the node where one of the RabbitMQ server Pods is deployed as `RABBIT_SERVER_NODE_IP`, and the Pod's internal IP as `RABBIT_SERVER_POD_IP`. The RabbitMQ deployment command using the Helm Chart is shown below, where `rabbit.yaml` specifies the cluster configurations:

```
helm install rabbitmq bitnami/rabbitmq -namespace stf008 -f rabbit.yaml
```

The RabbitMQ service can be uninstalled with:

```
helm uninstall rabbitmq -n stf008
```

### 5.2.2 Start Consumer Application

Since the producer and consumer applications communicate through a RabbitMQ streaming service, we use a Golang-based RabbitMQ consumer application that connects over either TLS or non-TLS, declares a queue (`MSG_QUEUE`), and consumes messages. The consumer application utilizes the amqp091-go (version 1.10.0) [38] RabbitMQ AMQP client library to implement RabbitMQ APIs. The application is started with:

```
go run consumer-app.go
```

The consumer application uses `amqp.DialConfig("amqp://username:password@RABBIT_SERVER_NODE_IP:30672")` API from the amqp091-go client library to establish a connection with the RabbitMQ server. Here, port `30672` is the NodePort mapped to the RabbitMQ server's non-TLS port `5672`. If the consumer application needs to use TLS, it can instead connect through port `30671`, which is mapped to the RabbitMQ TLS port `5671`.

### 5.2.3 Set up and Start of Producer and Consumer S2CS

The setup and startup of the producer and consumer S2CS are the same as described in §5.1.2 and §5.1.3, except that we modify their Pods' YAML files. Specifically, the Services `prod-s2cs-service` and `cons-s2cs-service` are updated to expose the Pods' ports `5000` and `5100-5110` externally through NodePorts `30500`, `30510-30520` and `30700`, `30710-30720` on the host nodes where the Pods are deployed. Let us refer to the node and Pod IPs of the Pods where the producer and consumer S2CS are deployed as `PROD_S2CS_NODE_IP`, `CONS_S2CS_NODE_IP`, `PROD_S2CS_POD_IP`, and `CONS_S2CS_POD_IP`, respectively.

### 5.2.4 Start S2UC and Send Requests

To start S2UC, we used a login node on the Andes cluster, which only supports Apptainer [1] containers. The first step is to pull the SciStream Docker image (version 1.2.1) from Docker Hub using the apptainer pull command, which generates a `scistream_1.2.1.sif` file locally. The Apptainer container can then be started with:

```
apptainer exec --bind ./certs:/certs scistream_1.2.1.sif /bin/bash
```

The certificates generated by the producer and consumer S2CS have been copied to the `./certs` directory so they can be shared with the S2UC for connection authentication. The container then binds the local `./certs` directory to `/certs` inside the container.

1. Send the inbound request from the container using S2UC:

   ```
   s2uc inbound-request --server_cert=certs/cons-server.crt --remote_ip
   RABBIT_SERVER_POD_IP --s2cs CONS_S2CS_NODE_IP:30700 --receiver_ports
   5672 --num_conn 1
   ```

   The request is sent to the consumer S2CS at its node IP (`CONS_S2CS_NODE_IP`) and NodePort `30700`, which maps to port `5000` inside the Pod. From the consumer S2CS, the next hop (`remote_ip`) is the Pod IP of one of the RabbitMQ servers deployed on a DSN. The receiver port is the non-TLS port `5672`. Note that NodePort `30672` does not need to be used here, since both the consumer S2CS and the RabbitMQ server run inside the same Olivine cluster and can communicate directly without NodePorts. As a result of this inbound request, the consumer S2CS creates a consumer S2DS proxy (`PROXY_C`) and assigns it a unique ID (`UID`). We will refer to the NodePort mapped to this `PROXY_C` as `PROXY_C_NODEPORT`.

2. Send the outbound request from the container using S2UC:

   ```
   s2uc outbound-request --server_cert=certs/prod-server.crt
   --remote_ip CONS_S2CS_NODE_IP --s2cs PROD_S2CS_NODE_IP:30500
   --receiver_ports PROXY_C_NODEPORT --num_conn 1 UID
   CONS_S2CS_NODE_IP:PROXY_C_NODEPORT
   ```

Here, the producer S2CS does not need to use the consumer S2CS node IP (`CONS_S2CS_NODE_IP`) to communicate, since both are in the same cluster and the consumer S2CS Pod IP could be used directly. However, we still use `CONS_S2CS_NODE_IP` to mirror a real cross-facility or cross-cluster streaming scenario, where the producer S2CS Pod would be deployed in a different cluster than the consumer S2CS. As a result of this outbound request, the producer S2CS creates an S2DS proxy (`PROXY_P`). We will refer to the NodePort mapped to this `PROXY_P` as `PROXY_P_NODEPORT`.

#### 5.2.5 Stream Data

To send data to the consumer application through the streaming service, we use a Go-based RabbitMQ producer application. This producer sends messages to the queue `MSG_QUEUE` in the RabbitMQ server and can be started with:

```
go run producer-app.go
```

The producer uses the same API to connect to the RabbitMQ streaming service but specifies a different URL: `amqp.DialConfig("amqp://username:password@PROD_S2CS_NODE_IP:PROXY_P_NODEPORT")`. Here, `PROD_S2CS_NODE_IP` is the producer S2CS node IP, and `PROXY_P_NODEPORT` is the NodePort mapped to the proxy opened by the producer S2CS as a result of the outbound request. We use AMQP without TLS in this setup because SciStream's tunneling through proxies already provides encryption and authentication via its TLS layer. In a true cross-facility deployment, the only externally exposed path is this tunnel, ensuring that all traffic is protected by SciStream's TLS-secured channel. Once the data reaches the RabbitMQ server inside the HPC facility, it is already encrypted, so additional TLS between the consumer S2CS and RabbitMQ is unnecessary.

## 6. EVALUATION

In this Section, we present the evaluation of SciStream on ACE specifically utilizing setup 2 described in §5.2 since this setup most closely match with real streaming scenario setup. We first describe the streaming workloads and a streaming simulator developed for performing the evaluation and then we measure the throughput and latency for the streaming workloads when streamed from producer to consumer application through SciStream.

### 6.1 STREAMING WORKLOADS

We define streaming workloads based on two IRI science workflows, GRETA/Deleria [13] and SLAC-LCLS [40] selected from the OLCF Science Pilots and Workflows initiative [36]. This initiative aims to implement them on the ACE infrastructure to support experimental steering and cross-facility integration across diverse scientific domains.

GRETA is a gamma-ray spectrometer currently being deployed at the Facility for Rare Isotope Beams at Michigan State University. It enables real-time analysis of gamma-ray energy and 3D position with up to 100x greater sensitivity than existing detectors. The associated workflow software, Deleria, continuously streams experimental data over ESNet to hundreds of analysis processes on an HPC system, processing up to 500K events per second. Deleria supports time-sensitive streaming and has been deployed across ESNet and ACE to demonstrate a distributed experimental pipeline. Recent emulation experiments on ACE scaled to 120 simulated detectors, achieving sustained bi-directional streaming rates of ~35 Gb/s.

The LCLS at SLAC National Accelerator Laboratory provides X-ray scattering for molecular structure analysis and streams experimental data to enable rapid analysis and decision-making between experiment runs. The LCLStream pilot project trains a generalist AI model using streamed detector data, from both archived and live LCLS/LCLS-II [39] experiments, to support tasks like hit classification, Bragg peak segmentation, and image reconstruction. This AI-driven approach serves as a shared backbone for various downstream data analysis tasks. With the new LCLS-II producing data at 400x the rate of its predecessor, streaming up to 100 GB/s to HPC systems will be essential for responsive analysis and experiment steering. LCLStream aims to support online streaming and real-time analysis during experiment execution, eliminating delays associated with waiting for data to be written to file storage systems before processing.

**Table 1. Data streaming characteristics for streaming workloads - Deleria and LCLS.**

| Characteristics | Deleria | LCLS |
|---|---|---|
| Payload size | ≈KiB range | ≈1 MiB |
| Payload format | Binary | HDF5 |
| Payload element | Events | Events |
| Data packaging | Variable # events/msg | Variable # events/msg |
| Data rate | 32 Gbps | 30 Gbps |
| Consumption parallelism | Parallel (non-MPI) | Parallel (MPI-based) |
| Production parallelism | Parallel (non-MPI) | Parallel (MPI-based) |

Table 1 shows the key data streaming characteristics relevant to both workloads. The LCLS stream uses ≈1 MiB data payloads with a steady data rate of ≈30 Gbps sustained over 1–100 minutes. Each message contains an HDF5-formatted file, with producers and consumers launched using MPI. Messages are pushed to consumers in a round-robin fashion as they become available in the queue.

In contrast, Deleria streams messages in the KiB range, each containing multiple experimental events batched together. The number of events per message is variable. Data messages use a compressed binary format, while control messages are encoded in JSON. Depending on the type of experiment, the GRETA detector sustains a steady data rate of up to 32 Gbps once an experiment begins. Producers and consumers do not use MPI. Instead, consumers pull event batches asynchronously from a remote forward buffer, while pushing processed events to a remote event builder. Although the Deleria workload's payload size is variable in the KiB range and streams a variable number of events per message, for consistency, we fix the payload size to 2 KiB per event and the number of events per message to eight, resulting in a 16 KiB message size.

To indicate that these workloads are synthetic streaming workloads derived from Deleria and LCLS, we refer to them as **Dstream** and **Lstream**, respectively.

### 6.2 STREAMING SIMULATOR

To simulate the streaming experiments, we developed a Golang-based simulator[1] and utilized the amqp091-go (version 1.10.0) [38] RabbitMQ AMQP client library to implement RabbitMQ APIs. The simulator accepts the streaming characteristics of workflows, as listed in Table 1. Additionally, the simulator allows specifying RabbiMQ specific parameters (e.g., type of acknowledgements, number of queues, prefetch count), experiment configurations (e.g., number of producers and consumers, message count, experiment duration), and SciStream specific options (e.g., URL for connection, number of connections, TLS). For a given message count or test duration, the simulator runs the experiment with the specified number of producers and consumers. Each producer is identical in function and is responsible for generating workload based on the input workload characteristics and sending data to the RabbitMQ server through SciStream according to the specified parameters. Similarly, each consumer is identical and is designed to receive messages from the RabbitMQ server based on the same set of parameters.

In addition to the producers and consumers, the simulator includes a coordinator component that serves two primary functions. First, it informs producers and consumers about which message queues to use. Second, it collects metrics from individual consumers/producers and reports the aggregate results for the entire experiment. Each component, upon startup, handles the initialization of all required queues for the experiment run. The simulator supports launching both MPI-based and non-MPI producers and consumers.

### 6.3 EVALUATION METRICS AND SETUP

For the simulations presented, we measured two metrics: throughput and Round-trip time (RTT). Throughput refers to the aggregate message rate (messages per second) from all consumers involved in each experiment. RTT is the time it takes for a message to travel from a producer to a consumer and for the corresponding reply to return to the producer. Each data point represents the average of three runs, with each run streaming up to 128K messages. All tests were performed with an equal number of producers and consumers to evaluate scaling behavior, and in every case the consumers were started before the producers.

To run the simulations, a total of 33 nodes from OLCF's Andes system [32] were used: 16 nodes for producers, 16 nodes for consumers, and 1 node for the coordinator.

### 6.4 MESSAGING PARAMETERS

To align streaming behavior with the workload characteristics shown in Table 1, we configure RabbitMQ with specific parameters. For both workloads, to measure throughput, we adopt the work queue model,

[1]Our data streaming simulator, configuration files, and Pod definitions are publicly available here: https://github.com/Ann-Geo/StreamSim/

where producers send messages to shared queues and messages are distributed nearly evenly across multiple consumers. Where as to measure the RTT, we use the same work queue model for request messages, but employ the direct routing model for replies. Each producer has a dedicated reply queue, ensuring that replies are routed back to the correct producer. This prevents misrouting and eliminates the risk of a producer waiting indefinitely for a reply intended for it but consumed by another. For both the above patterns we used two shared work queues to achieve increased throughput [23].

All queue models use RabbitMQ's classic queues, which retain a fixed number of messages in memory and support configurable durability. We set the queue overflow policy to "reject-publish", allowing producers to detect backpressure, handle rejected messages, and attempt republishing. Of the total RAM allocated to RabbitMQ servers, 80% is reserved for data payload queues, with the remaining 20% allocated to additional queues used for control messages, and workflow simulation management. Additionally we enable batch wise producer and consumer acknowledgements for guaranteed message reception.

### 6.5 THROUGHPUT MEASUREMENTS

Figure 6 shows the aggregate throughput for both workloads. We also evaluated two additional SciStream tuning options: network proxy type and number of connections to proxies. Specifically, we tested Stunnel and HAProxy proxies, with up to four connections used in the HAProxy configuration. For the Dstream workload with a single producer and consumer, configuration with HAProxy achieved the highest throughput at 6.3K msgs/sec, while other configurations ranged between 4.4K and 6.2K msgs/sec.

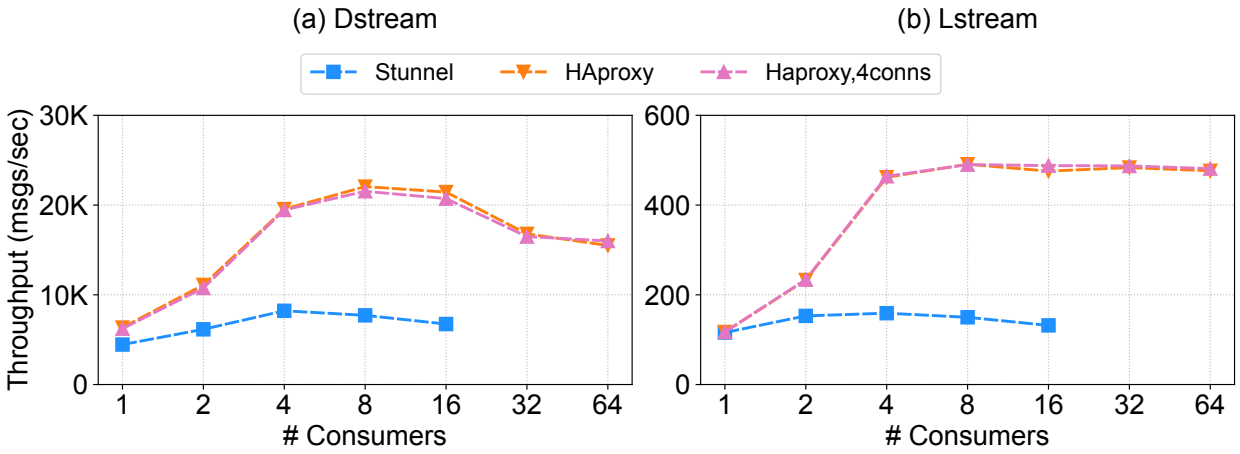


**Figure 6. Throughput (msgs/sec) for (a) Dstream and (b) Lstream workloads.**

In the configuration using Stunnel, throughput showed no significant improvement beyond a single consumer. Moreover, Stunnel could support a maximum of 16 simultaneous connections in our setup, making configurations with 32 and 64 consumers infeasible (no data points shown). This limitation is due to Stunnel's design, which favors a few long-lived, TLS-wrapped flows rather than load balancing. By contrast, configuration with HAProxy scaled better, reaching up to 19K msgs/sec with four consumers, but throughput stagnated and began to decline beyond eight consumers. Increasing proxy connections to four showed no significant performance gain.

For the Lstream workload, which uses a larger payload size, configuration with HAProxy scaled well up to four consumers, after which throughput plateaued. Stunnel showed similar limitations as with Dstream.

### 6.6 ROUND TRIP TIME MEASUREMENTS

We measured the per-message RTT, the time taken for a message to travel from a producer to a consumer and return to the producer. Figure 7 shows the median RTT for both workloads. Since the Stunnel configuration showed poor performance in earlier tests, we excluded it from further RTT evaluations and focused only on HAProxy type.

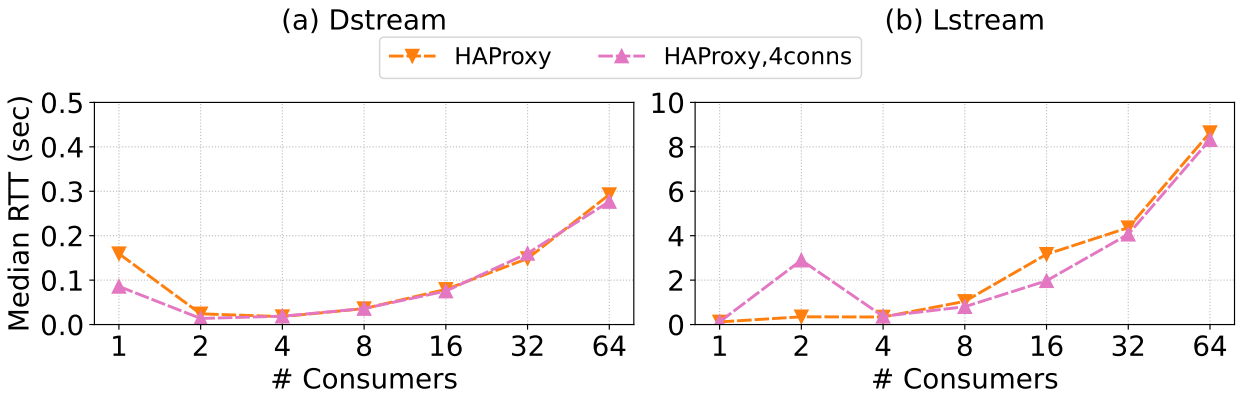


**Figure 7. Median RTT (sec) for (a) Dstream and (b) Lstream workloads.**

For Dstream workload, median RTTs were maintained under 0.5 seconds across all consumer counts. The HAProxy configuration achieved a minimum of 17ms with four consumers. RTT increased slightly as the number of consumers grew beyond eight. In the Lstream workload, for a single consumer, RTTs were under 200ms. Median RTTs remained below 10 seconds across all consumer counts.

Since median latency cannot capture per-message RTT variations, we present the CDF of RTTs for all messages in Figure 8. The RTT distributions remain consistently tight, with limited variation. Notably, in the 64-consumer case, 80% of message RTTs are under 0.7 seconds for Dstream and 12.5 seconds for Lstream, demonstrating uniformity in latency. However, increasing the number of connections from one to four does not produce observable improvements in RTT.

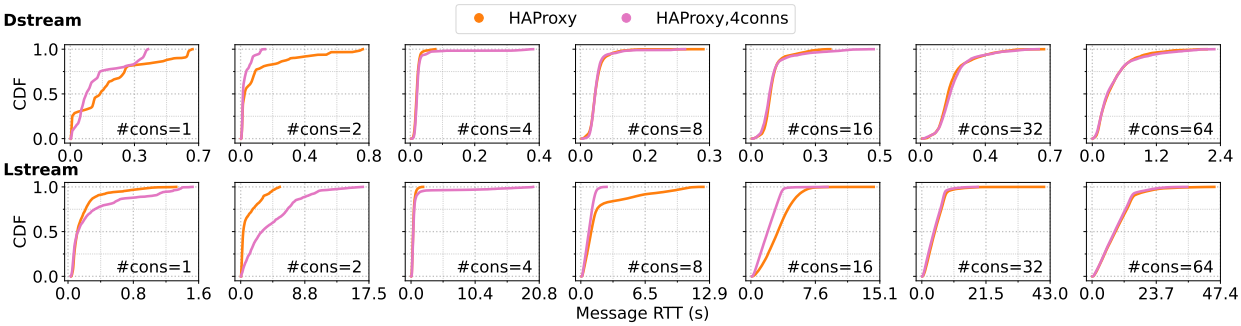


**Figure 8. CDF of individual message RTTs for Dstream and Lstream workloads, with the number of consumers varying from 1 to 64.**

## 7. CONCLUSION, CHALLENGES, AND FUTURE WORK

In this report, we described the steps taken to deploy and evaluate SciStream on OLCF's ACE infrastructure. We first prepared the SciStream components for deployment on the OpenShift environment of the Olivine cluster. The deployment was then evaluated using synthetic streaming workloads derived from IRI science workflows, along with a streaming simulator in which producer and consumer applications ran on compute nodes and communicated through SciStream with the support of a RabbitMQ streaming service. During deployment and evaluation, we encountered multiple challenges that also revealed opportunities for future extensions and improvements. We describe these challenges and possible directions below.

**Usage of high-speed network**: The evaluation shows that throughput eventually stagnates as the number of consumers increases. A primary reason is the 1 Gbps network link between the compute nodes (producers/consumers) and the DSNs (RabbitMQ servers). While the DSNs have 100 Gbps interfaces, configuration issues within our OpenShift environment have prevented their effective use. Ongoing efforts are focused on reworking the DSNs to make the 100 Gbps interfaces fully usable [24].

**Secure access via OpenShift Routes**: SciStream requires clients to connect to a reachable IP and port. OLCF OpenShift clusters can serve applications on NodePorts, but security constraints generally require that external access (traffic from outside OLCF) use TLS-encrypted connections through OpenShift Routes [22] (and Istio gateways [3]) on port 443, so that traffic can be inspected. While it is possible to configure OpenShift to support Routes on ports other than 443, this is unlikely from the platform's perspective because it would be complex.

**Routing constraints**: At OLCF, routing on port 443 is hostname-based, whereas SciStream identifies connections by port numbers or Unique identifiers (UIDs). Because Route traffic can only come in on port 443, incoming traffic is generally differentiated by hostname rather than port, which conflicts with SciStream's port-based architecture (though hostname-based routing may be possible). Services like S3M [37, 19] deployed at OLCF already uses hostname-based routing to address this limitation. For compatibility, SciStream would need new features to support or adapt to hostname-based routing.

**NodePort limitations**: While exceptions can be requested to allow external access via NodePorts, this requires additional firewall rules or policies. Reserving a NodePort range for SciStream's dynamic ports would help, but this feature is only supported in newer Kubernetes versions. In addition, there may be policies implemented later to disallow NodePorts on OLCF clusters altogether, as traffic through them cannot be easily inspected. Given this uncertainty, building SciStream to depend heavily on NodePorts could be possible only when future policies and Istio-based routing are in place.

## ACKNOWLEDGEMENT

We thank our colleagues Michael Brim, Christopher Zimmer, and Sarp Oral for their review, guidance, and suggestions; Nick Schmitt, Ethan O'Dell, Steven Lu, Zach Mayes, and A.J. Ruckman, for their help in resolving technical challenges during the deployment of SciStream; and Tyler Skluzacek, Paul Bryant, Brian Etz, David Rogers, Gustav Jansen, Ross Miller, and Subil Abraham for providing valuable information and clarifications. We also thank the SciStream team at ANL, Flavio Castro and Rajkumar Kettimuthu for helping us better understand SciStream's capabilities. This research used resources of the Oak Ridge Leadership Computing Facility located at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under contract No. DE-AC05-00OR22725.

## 8. REFERENCES


[1] Contributors to the Apptainer project. *Apptainer, Empower your Applications*. 2025. URL: https://apptainer.org/.

[2] Scott Atchley et al. "Frontier: Exploring Exascale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23. Denver, CO, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: 10.1145/3581784.3607089. URL: https://doi.org/10.1145/3581784.3607089.

[3] Istio Authors. *Istio, Service Mesh Simplified*. 2025. URL: https://istio.io/.

[4] The Kubernetes Authors. *Kubernetes Service*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[5] The ZeroMQ authors. *ZeroMQ, An open-source universal messaging library*. 2025. URL: https://zeromq.org/.

[6] Anakha V Babu et al. "Deep learning at the edge enables real-time streaming ptychographic imaging". In: *Nature Communications* 14.1 (2023), p. 7059.

[7] US Department of Energy Office of Science Berkeley Lab. *ESnet, Energy Sciences Network*. 2025. URL: https://www.es.net/.

[8] Michael J. Brim et al. *A High-level Design for Bidirectional Data Streaming to High-Performance Computing Systems from External Science Facilities*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), Mar. 2024. DOI: 10.2172/2338264. URL: https://www.osti.gov/biblio/2338264.

[9] Broadcom. *AMQP 0-9-1 Model Explained*. 2025. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.

[10] Broadcom. *RabbitMQ 4.1 Documentation*. 2025. URL: https://www.rabbitmq.com/docs.

[11] Kyle Chard, Steven Tuecke, and Ian Foster. "Globus: Recent enhancements and future plans". In: *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*. 2016, pp. 1–8.

[12] Joaquin Chung et al. "SciStream: Architecture and Toolkit for Data Streaming between Federated Science Instruments". In: HPDC '22. Minneapolis, MN, USA: Association for Computing Machinery, 2022, pp. 185–198. ISBN: 9781450391993. DOI: 10.1145/3502181.3531475. URL: https://doi.org/10.1145/3502181.3531475.

[13] Cromaz, Mario et al. "Simple and Scalable Streaming: The GRETA Data Pipeline*". In: *EPJ Web Conf.* 251 (2021), p. 04018. DOI: 10.1051/epjconf/202125104018. URL: https://doi.org/10.1051/epjconf/202125104018.

[14] Eli Dart et al. "The Science DMZ: a network design pattern for data-intensive science". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503245. URL: https://doi.org/10.1145/2503210.2503245.

[15] Ewa Deelman et al. "Pegasus, a workflow management system for science automation". In: *Future Generation Computer Systems* 46 (2015), pp. 17–35.

[16] SciStream developers. *Scistream*. 2025. URL: https://github.com/scistream/scistream-proto.

[17] SciStream developers. *SciStream SuperComputing 24 Hands-on Tutorial*. 2025. URL: https://github.com/scistream/sc24-tutorial.

[18] SciStream developers. *Welcome to SciStream Documentation*. 2025. URL: https://scistream.readthedocs.io/en/latest/.

[19] OLCF Docs. *Get Started with OLCF API*. 2025. URL: https://s3m.apps.olivine.ccs.ornl.gov/docs/gen/getting-started.html.
[20] Apache Software Foundation. *Apache Kafka*. 2025. URL: https://kafka.apache.org/.
[21] The Linux Foundation. *Bitnami package for RabbitMQ*. 2025. URL: https://artifacthub.io/packages/helm/bitnami/rabbitmq.
[22] The Linux Foundation. *Kubernetes Ingress*. 2025. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.
[23] Anjus George et al. *A Study on Messaging Trade-offs in Data Streaming for Scientific Workflows*. 2025. arXiv: 2509.07199 [cs.DC]. URL: https://arxiv.org/abs/2509.07199.
[24] Anjus George et al. "From Edge to HPC: Investigating Cross-Facility Data Streaming Architectures". In: *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops '25. Association for Computing Machinery, 2025, pp. 949–959. ISBN: 9798400718717. DOI: 10.1145/3731599.3767452. URL: https://doi.org/10.1145/3731599.3767452.
[25] Jeremy Goecks et al. "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences". In: *Genome biology* 11.8 (2010), R86.
[26] The IRI Management Council's Leadership Group. *Integrated Research Infrastructure*. 2025. URL: https://iri.science/.
[27] Red Hat. *Red Hat OpenShift*. 2025. URL: https://developers.redhat.com/products/openshift/overview.
[28] Eric Jackson Hobbit. *nc(1) - Linux man page*. 2006. URL: http://linux.die.net/man/1/nc.
[29] Docker Inc. *Docker for AI. Compose. Build. Deploy.* 2025. URL: https://hub.docker.com/.
[30] Docker Inc. *Docker, Develop Faster*. 2025. URL: https://www.docker.com/.
[31] William L. Miller et al. "Integrated Research Infrastructure Architecture Blueprint Activity (Final Report 2023)". In: (July 2023). DOI: 10.2172/1984466.
[32] OLCF. *Andes User Guide*. 2025. URL: https://docs.olcf.ornl.gov/systems/andes_user_guide.html.
[33] OLCF. *Odo, System Overview*. 2025. URL: https://docs.olcf.ornl.gov/systems/odo_user_guide.html.
[34] OLCF. *Slate, Overview*. 2025. URL: https://docs.olcf.ornl.gov/services_and_applications/slate/overview.html.
[35] OLCF. *Standing up the nation's supercomputing pipeline for streaming big data in real time*. 2024. URL: https://www.ornl.gov/news/standing-nations-supercomputing-pipeline-streaming-big-data-real-time.
[36] Sarp Oral et al. *OLCF's Advanced Computing Ecosystem (ACE): FY24 Efforts for the DOE Integrated Research Infrastructure (IRI) Program*. Tech. rep. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), Nov. 2024. DOI: 10.2172/2477506. URL: https://www.osti.gov/biblio/2477506.
[37] Tyler J. Skluzacek et al. "Secure API-Driven Research Automation to Accelerate Scientific Discovery". In: *Practice and Experience in Advanced Research Computing 2025: The Power of Collaboration*. PEARC '25. Association for Computing Machinery, 2025. ISBN: 9798400713989. DOI: 10.1145/3708035.3736072. URL: https://doi.org/10.1145/3708035.3736072.
[38] RabbitMQ core team. *Go RabbitMQ Client Library*. 2025. URL: https://github.com/rabbitmq/amqp091-go/tree/main.
[39] Jana Thayer et al. "Massive Scale Data Analytics at LCLS-II". In: *EPJ Web of Conf.* 295 (2024), p. 13002. DOI: 10.1051/epjconf/202429513002. URL: https://doi.org/10.1051/epjconf/202429513002.

[40] Cong Wang et al. “End-to-end deep learning pipeline for real-time Bragg peak segmentation: from training to large-scale deployment”. In: *Frontiers in High Performance Computing* Volume 3 - 2025 (2025). ISSN: 2813-7337. DOI: 10.3389/fhpcp.2025.1536471. URL: https://www.frontiersin.org/journals/high-performance-computing/articles/10.3389/fhpcp.2025.1536471.

[41] Von Welch et al. “A roadmap for using NSF cyberinfrastructure with InCommon”. In: *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. 2011, pp. 1–2.

[42] Justin M Wozniak et al. “Swift/t: Large-scale application composition via distributed-memory dataflow processing”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 95–102.

[43] Junqi Yin et al. “Integrated edge-to-exascale workflow for real-time steering in neutron scattering experiments”. In: *Structural Dynamics* 11.6 (Dec. 2024), p. 064303. ISSN: 2329-7778. DOI: 10.1063/4.0000279. eprint: https://pubs.aip.org/aca/sdy/article-pdf/doi/10.1063/4.0000279/20319498/064303_1_4.0000279.pdf. URL: https://doi.org/10.1063/4.0000279.