

Seminar Report

Building an End-to-End Edge-to-HPC Streaming Pipeline

Jan Göwecke

MatrNr: 21674346

Supervisor: Jonathan Decker, Sepehr Mahmoodian

Georg-August-Universität Göttingen
Institute of Computer Science

October 19, 2026

Abstract

Scientific instruments and sensors produce continuous data streams whose analysis requires the computing capacity of high-performance computing (HPC) systems. HPC, however, is built around batch scheduling: compute nodes are not reachable from outside, services end with their job, and existing streaming solutions depend on infrastructure operated by the facility. This report examines how such streams can be connected to HPC with the rights of a regular user, and which latency and reliability this path achieves. A three-tier prototype was built at the GWDG data center: simulated edge devices publish via MQTT to a cloud VM, from where a bridge forwards the data through an SSH tunnel via the login node to Apache Kafka inside a Slurm job. Instrumented experiments with broker-side timestamps decompose the latency into segments and include a load ramp on real infrastructure. The path from cloud to HPC adds a median latency of about 6–9 ms at up to 5,000 messages per second, without loss in fault-free operation, while the wide-area link dominates the estimated end-to-end latency of about 20 ms. The results indicate that the bottleneck of such a pipeline lies not in transport or data processing but in establishing and operating connectivity across tiers: the permanent operation of a user-held tunnel is a policy question, and data do not survive the end of a job.

I hereby declare that all uses of tools and AI as well as assistance from third parties in this work have been fully and accurately disclosed in the Transparency Report in Appendix Section C.

Missing or incorrect information may be considered an attempt to cheat.

Contents

List of Tables	iv
List of Figures	iv
List of Abbreviations	v
1 Introduction	1
2 Background	2
2.1 HPC Access Constraints and SSH Port Forwarding	2
2.2 MQTT, Apache Kafka and Delivery Guarantees	2
2.3 Cross-Facility Streaming Architectures	3
2.4 Measuring Latency and Throughput Across Machines	4
3 Related Work	4
3.1 Facility-Provided Streaming Infrastructure	4
3.2 Message Brokers on HPC and Across the Continuum	4
3.3 User-Side Access to Services on Compute Nodes	5
3.4 Evaluation Approaches and Research Gap	5
4 Methodology	6
4.1 Research Questions and Approach	6
4.2 Evaluation Criteria	7
4.3 Setup and Measurement Design	7
4.4 Experiments and Analysis	9
4.5 Deviations and Assumptions	10
5 Results	10
5.1 Architecture and Requirement Fulfilment	10
5.2 Implementation Findings	11
5.3 Functional Validation	12
5.4 Latency Breakdown on Real Infrastructure	12
5.5 Behavior under Load	13
5.6 Reliability and Fault Behavior	15
5.7 Answers to the Research Questions	16
6 Discussion	17
6.1 Interpretation	17
6.2 Practical Implications	17
6.3 Limitations and Open Questions	18
7 Conclusion	18
References	20
A Test Environment and Run Details	A1

B	Code	C4
C	Transparency Report: Tools and Third-Party Contributions	C4
C.1	Overview	C4
C.2	Per Section	C5

List of Tables

1	Comparison of MQTT and Kafka	3
2	Comparison of related approaches.	6
3	Requirements for RQ1.	7
4	Hypotheses for RQ2 and decision rules.	8
5	Measured segments.	9
6	Experiments.	9
7	Deviations from the project plan.	10
8	Options for connecting the cloud VM to the High-Performance Computing (HPC) broker.	11
9	Fulfilment of requirements R1–R7	12
10	Load ramp: load, data rate, loss and per-segment latency	14
11	Results with respect to research questions and hypotheses.	16
12	Test environment.	A1
13	Run details.	A2
14	Selected implementation obstacles.	A3

List of Figures

1	System under test with measurement points t_0 – t_5	8
2	Segment medians at 100 msg/s	13
3	Cloud-to-HPC latency versus offered load	15

List of Abbreviations

DMZ Demilitarized Zone

DTS Direct Streaming

HPC High-Performance Computing

IoT Internet of Things

MQTT Message Queuing Telemetry Transport

MSS Managed Service Streaming

NTP Network Time Protocol

PRS Proxied Streaming

QoS Quality of Service

RTT Round-Trip Time

SNTP Simple Network Time Protocol

SSH Secure Shell

TCP Transmission Control Protocol

TLS Transport Layer Security

VM Virtual Machine

WAN Wide-Area Network

1 Introduction

Scientific instruments generate data at rates that require online processing, while the computing capacity available at the source is limited and moving data via the file system introduces latencies that are unacceptable for such processing [Chu+22]. Similarly, emerging cyber-physical systems require low-latency processing of streaming sensor data, while the computationally intensive tasks they depend on are executed on remote High-Performance Computing (HPC) systems under batch scheduling [Kur+26]. Making HPC capacity available to continuous data streams in a timely manner is therefore a recurring requirement.

This requirement conflicts with the operating model of HPC systems. Any service on a compute node is part of a batch job and therefore bounded by its time limit; compute nodes are not addressable from outside, and external access is restricted to Secure Shell (SSH) connections to the login nodes [Doo+26]. In addition, the node that hosts a consumer is only known once the scheduler has allocated the job. Existing approaches overcome these constraints with infrastructure that is provided and operated by the facility, such as proxies on gateway nodes in a Science DMZ [Chu+22] or facility-operated gateway nodes and managed streaming services [Geo+25]. Even then, network policy can hinder their external use [Geo+25]. For regular users without administrative rights, these routes are hardly accessible.

Two gaps result from this situation (Section 3.4). No approach connects continuous edge data streams to a broker inside a running batch job using only the rights of a regular user, and no study provides a segment-resolved characterization of the latency and reliability of such a path on real edge, cloud and HPC infrastructure. Accordingly, this report addresses two research questions:

- RQ1 (constructive): How can a complete edge-to-HPC streaming pipeline be built with open-source tools under real data-center conditions?
- RQ2 (empirical): Which latency and reliability properties does this pipeline exhibit under varying load?

To answer them, a three-tier prototype was built on the infrastructure of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG). Simulated edge devices publish readings via Message Queuing Telemetry Transport (MQTT) to a cloud VM, from where a bridge forwards them through an SSH local forward via the login node to Apache Kafka inside a Slurm job. The prototype is assessed against explicit requirements (RQ1) and characterized in instrumented experiments on real infrastructure, including a load ramp up to 5,000 msg/s (RQ2). The work focuses on the transport and integration problem; production hardening, real sensor hardware, domain-specific processing and multi-site operation are out of scope.

This report makes three contributions:

1. A user-level instantiation of proxied streaming (PRS, Section 2.3), built with open-source tools and the rights of a regular user and assessed against explicit requirements, including the policy limits of its permanent operation.
2. A segment-wise measurement design that uses broker-side timestamps to separate latency segments measured on a single clock from those spanning two clocks.

3. An empirical characterization on real edge, cloud and HPC infrastructure. It indicates that the path from cloud to HPC adds only a few milliseconds at loads up to 5,000 msg/s, and that the main obstacles lie in establishing and operating connectivity across tiers rather than in transport or data processing.

For users, the results show what can be achieved with regular rights; for operators, they indicate which sanctioned options would enable robust operation.

The remainder of this report is structured as follows. Section 2 introduces the background and Section 3 reviews related work. Section 4 describes the methodology, Section 5 presents the results, Section 6 discusses them, and Section 7 concludes.

2 Background

This section introduces the concepts on which the design and evaluation of the pipeline build. Familiarity with the basic operating model of HPC systems, i.e., batch scheduling with Slurm [YJG03], login and compute nodes, and walltime limits, is assumed. Section 2.1 describes the access constraints between external data sources and compute nodes, Section 2.2 the two messaging systems used in the pipeline and their delivery guarantees, Section 2.3 a taxonomy of cross-facility streaming architectures that serves as the analytical frame for Section 5 and Section 2.4 the evaluation metrics and the limits of measuring them across machines. Concrete systems and prior studies are reviewed in Section 3.

2.1 HPC Access Constraints and SSH Port Forwarding

HPC centers shield their compute nodes within an internal network: compute nodes are neither addressable from outside nor able to access the internet directly, and external access is restricted to SSH connections to the login nodes [Doo+26]. Any service on a compute node is part of a batch job and therefore bounded by its time limit [Doo+26]. For streaming, this means that an external producer cannot connect to a consumer on a compute node directly, and that the node hosting the consumer is only known once the scheduler has allocated the job.

SSH port forwarding carries Transmission Control Protocol (TCP) connections through an existing SSH connection [YL06]. With local forwarding (`ssh -L`), the client listens on a local port and the server opens the connection to a target reachable from its own network; remote forwarding (`ssh -R`) works in the opposite direction [YL06]. In the HPC context, the direction of connection setup is decisive: a local forward can be initiated from outside towards the login node, which reaches compute nodes internally, whereas a remote forward originating from a compute node would require an outbound connection from inside the cluster.

2.2 MQTT, Apache Kafka and Delivery Guarantees

MQTT and Kafka are both publish/subscribe systems in which a broker decouples producers from consumers, but they follow different design goals (Table 1). MQTT is a lightweight protocol widely used in Internet of Things (IoT) systems [Nai17]: clients publish messages to hierarchical topics, and the broker forwards them to all clients with a

matching subscription [BG14]. Its three quality-of-service levels, at most once (QoS 0), at least once (QoS 1) and exactly once (QoS 2), apply to the exchange between one client and the broker; messages for a temporarily disconnected subscriber are only retained if it uses a persistent session [BG14].

Kafka was developed to collect and deliver high volumes of log data with low latency [KNR11]. Records are appended to topics divided into partitions; each partition is an ordered log in which a record is identified by its offset [HMU20; Wan+21]. Consumers pull records and track their progress by committing offsets, either automatically or explicitly, so that they can resume from the last committed position as long as the records are retained [HMU20]. The distance between the latest offset of a partition and a consumer’s position is referred to as consumer lag. Producers collect records into batches to reduce the number of requests, which trades latency for throughput [HMU20].

Table 1: Comparison of MQTT and Kafka [BG14; KNR11; HMU20; Wan+21].

	MQTT	Kafka
Design goal	Lightweight messaging for constrained devices	High-volume, low-latency log delivery
Communication model	Broker pushes messages to subscribers	Consumers pull records from partitioned logs
Delivery guarantee	QoS 0/1/2 per client–broker hop	At-least-once by default
Persistence	Messages retained only for persistent sessions	Records retained for a configurable period, replay possible
Typical deployment	Edge devices, wide-area links	Data center backend

At-least-once delivery relies on retries, and since a sender that receives no acknowledgement cannot distinguish a lost message from a lost acknowledgement, retries may produce duplicates [Wan+21]. All these guarantees refer to a single hop between two components. When several systems are chained via a bridge, a message acknowledged on one hop is not automatically protected on the next. This reflects the end-to-end argument, according to which reliable delivery can only be fully ensured by the communication endpoints themselves; mechanisms on intermediate hops may improve performance but cannot replace an end-to-end check [SRC84].

2.3 Cross-Facility Streaming Architectures

George et al. [Geo+25] distinguish three architectures for streaming data from external producers to consumers in HPC jobs. In direct streaming (DTS), the streaming service runs on the destination nodes and is exposed through node-level ports, which yields the shortest path but requires administrators to route external traffic and open firewall rules. In proxied streaming (PRS), intermediary proxies relay the data; one proxy initiates the connection via known gateway nodes, so that administrators only need to pre-authorize stable endpoints rather than individual node addresses. In managed service streaming (MSS), the facility operates the streaming service behind a stable endpoint, and producers only require outbound connectivity [Geo+25]. The three architectures thus differ mainly in which party establishes and controls connectivity and in the resulting administrative effort. Section 5 uses this distinction to classify the architectural options of this project.

2.4 Measuring Latency and Throughput Across Machines

Event-time latency denotes the interval between the creation of a record and its output by the system, as opposed to processing-time latency, which only starts when the record is ingested [Kar+18]; it therefore reflects the delay perceived at the data source. In a multi-stage pipeline, it can be decomposed into segments between consecutive measurement points. A segment whose two timestamps are taken on the same machine is unaffected by clock differences, whereas a segment spanning two machines compares two clocks, each with its own offset from true time and its own resolution, whose uncertainties add up [Pax98]. Network Time Protocol (NTP) keeps clocks within a few milliseconds of reference time [Mil91], but does not guarantee synchronization on the short time scales of individual transfers [Pax98]. Round-trip times measured with a single clock avoid this problem, yet half the round-trip time is only a valid one-way estimate if the path is symmetric [Pax98].

The capacity of a streaming system is characterized by its sustainable throughput: the highest load it can handle without prolonged backpressure, i.e., without a continuously increasing event-time latency. Transient backpressure, after which latency returns to its previous level, does not violate this criterion [Kar+18].

3 Related Work

This section reviews approaches for connecting external data sources to HPC systems and remote back-ends. It examines how existing work establishes the data path (RQ1) and how it evaluates the resulting systems (RQ2), and derives the gap addressed by this report.

3.1 Facility-Provided Streaming Infrastructure

SciStream establishes memory-to-memory streams between instruments and remote HPC through on-demand transport-layer proxies on gateway nodes, which facilities deploy in their Science DMZ either as dedicated nodes or by repurposing existing data transfer nodes [Chu+22]. George et al. [Geo+25] implement the DTS, PRS and MSS architectures on facility-operated gateway nodes, realizing PRS with SciStream, and find that PRS performs comparably to DTS in most messaging patterns. Both approaches presuppose infrastructure that is provided and operated by the facility. Even with this infrastructure, George et al. [Geo+25] report that external use of PRS is hindered because external access to their cluster is restricted to HTTPS on port 443. Network policy rather than performance thus appears as the limiting factor, which this report revisits from the perspective of a regular user by instantiating PRS at user level: a user-operated cloud VM takes the role of the proxy, and the login node that of the known gateway.

3.2 Message Brokers on HPC and Across the Continuum

Pilot-Streaming deploys Kafka, Spark and Dask inside pilot jobs on HPC resources [LCJ18]. Its authors note that Kafka is not supported on HPC out of the box and requires manually written job scripts, and they stress that data sources are typically external [LCJ18]. Their evaluation, however, runs producers, brokers and processing on the same HPC

system, so the path by which external producers reach a broker on a compute node remains unaddressed. Pilot-Edge extends this approach to the edge-to-cloud continuum with Kafka-based brokering and envisages MQTT for low-power environments; it is evaluated on two cloud platforms, including a transatlantic deployment, and is limited to the edge and cloud layers [LRJ21]. CEFIoT likewise uses Kafka across edge and cloud, replicating data within local clusters and buffering it at the edge during connectivity loss until it can be forwarded [Jav+18], a store-and-forward capability that the bridge in this report’s pipeline lacks, as it buffers data only transiently in memory. Its fault tolerance is demonstrated functionally on five Raspberry Pi nodes, without latency or throughput measurements [Jav+18]. Neither approach addresses batch-scheduled HPC.

Kurafeeva et al. propose RBF (Reverse Backfill), which explicitly targets this setting [Kur+26]. It couples edge devices, a dedicated cluster and a shared HPC system through a fault-resilient distributed log and uses HPC jobs as opportunistic sources of model updates; files are exchanged via log repositories placed outside all firewalls, from which consumers fetch them [Kur+26]. Reported queue waits of 17–19 hours for CPU jobs [Kur+26] underline why batch scheduling is the central obstacle. RBF, however, sidesteps this obstacle through asynchrony: model updates arrive on a time scale of minutes to hours instead of data being streamed continuously into a running job. Its pull-based access moreover presupposes that jobs can connect outward to the repositories, which is limited at the facility used in this report [GWD26b]; Section 5 considers this pattern as an alternative architecture.

3.3 User-Side Access to Services on Compute Nodes

The GWDG documentation describes an SSH local forward, initiated from the user’s machine via a login node, to reach a service running in a Slurm job [GWD26a]. Private SSH keys are not permitted on the cluster, and network access of compute nodes is limited, so that connections to the outside world from within jobs should be avoided [GWD26b]. Doosthosseini et al. present SAIA, a Slurm-native platform for HPC-based services, which connects a cloud VM to Slurm jobs exclusively via SSH to the login node, using a key restricted by `ForceCommand` instead of port forwarding; it serves request–response traffic over per-request SSH sessions, with an average architectural overhead of about 42 ms [Doo+26]. Its authors regard port forwarding to compute nodes as a security risk and a policy violation at their center, and consider SSH keys stored on external servers a risk [Doo+26]. The documented tunnel is thus opened interactively by a user, whereas a streaming pipeline requires a persistent tunnel held by a server, which is the configuration SAIA’s operators assess as risky. In the prototype of this report, the tunnel is opened from the cloud VM but authenticated via SSH agent forwarding from the user’s machine, so that no private key is stored on the VM; this, however, ties the tunnel to an active user session and precludes unattended operation. Whether a user-level tunnel may be operated permanently is therefore a policy question rather than a technical one, which Section 5 takes up.

3.4 Evaluation Approaches and Research Gap

Table 2 contrasts the reviewed approaches. Methodologically, the evaluations are conducted within a single testbed or system with WAN (Wide-Area Network) conditions emulated or absent [Chu+22; Geo+25; LCJ18], restricted to cloud resources [LRJ21],

purely functional [Jav+18], or performed on real multi-site infrastructure at a time scale of minutes to hours [Kur+26]. Benchmarks of stream processing systems [Kar+18], whose latency and throughput definitions this report adopts (Section 2.4), and of Kafka, e.g., at about 420,000 messages per second on a single partition [HMU20], characterize components within a single data center and do not capture a multi-tier path.

Table 2: Comparison of related approaches.

Work	Data path	Facility-side prerequisite	Transfer mode	Evaluation setting
[Chu+22]	Proxies on gateway nodes	Gateway nodes in Science DMZ	Memory-to-memory stream	Testbed LAN, emulated WAN
[Geo+25]	DTS/PRS/MSS via gateway nodes	Facility gateway nodes, managed services	Message streaming	Within one HPC cluster
[LCJ18]	Kafka in pilot jobs	External access not addressed	Message streaming	Single HPC system
[LRJ21]	Kafka on edge and cloud resources	Cloud resources	Message streaming	Two clouds, transatlantic
[Jav+18]	Kafka clusters at edge and cloud	– (no HPC)	Replicated message streaming	Functional, five Raspberry Pi nodes
[Kur+26]	Logs outside firewalls, fetched by jobs	Repository reachable from HPC	Asynchronous files	Real multi-site deployment
[Doo+26]	SSH to login node, restricted key	Functional account, no admin changes	Request–response	Benchmarks and production
This work	MQTT to VM, SSH forward to Kafka in Slurm job	Regular user account + cloud VM with public IP	Continuous message stream	Real edge–cloud–HPC (low load); cloud–HPC load ramp; single runs

Within the reviewed literature, no approach connects continuous edge data streams to a broker inside a running batch job using only the rights of a regular user (RQ1), and no study provides a segment-resolved characterization of the latency and reliability of such a path on real edge, cloud and HPC infrastructure (RQ2). Section 4.2 derives the requirements for the system design from these gaps.

4 Methodology

This Section describes how the research questions were addressed, from the overall approach to the individual experiments, as implemented in the project repository; deviations from the initial project plan are stated in Section 4.5.

4.1 Research Questions and Approach

The work is guided by two research questions. RQ1 is constructive: *How can a complete edge-to-HPC streaming pipeline be built with open-source tools under real data-center conditions?* RQ2 is empirical: *Which latency and reliability properties does this pipeline exhibit under varying load?* RQ1 is addressed by building a prototype and assessing it against a set of requirements; RQ2 is addressed by instrumented experiments under controlled load. The prototype was developed in incremental, risk-oriented stages, each

validated before the next: local validation of the complete pipeline, the full chain on real infrastructure, a load ramp, and a fault injection. These stages correspond to the experiments in Table 6.

4.2 Evaluation Criteria

The requirements in Table 3 follow from the access constraints and delivery semantics described in Section 2 and from the gap identified in Section 3.4. They serve as criteria against which the architecture is assessed in Section 5.

Table 3: Requirements for RQ1.

ID	Requirement	Derived from
R1	Operation with the rights of a regular user, without administrative changes	Sec. 2.1, 3.3
R2	Adaptation to a target node that is only known at runtime	Sec. 2.1
R3	Robust transport from edge devices over a wide-area link	Sec. 2.2
R4	Decoupling and buffering before the HPC-side consumer	Sec. 2.2, 3.2
R5	Detectability of message loss and duplicates per source	Sec. 2.2
R6	Segment-wise measurability with minimal dependence on clock synchronization	Sec. 2.4
R7	Tolerance of the job time limit, i.e., resumption after a job ends without data loss	Sec. 2.1, 2.2

RQ2 is operationalized by three hypotheses (Table 4), which refer to the measurement points introduced in Section 4.3. H1 rests on the assumption that a wide-area link across the public internet contributes more delay than two data-center-internal links. H2 rests on the observation that the tunnel multiplexes all Kafka traffic over a single encrypted SSH connection relayed by a shared login node, so that encryption cost and head-of-line blocking within one TCP stream may limit throughput. H3 rests on the simulator’s separate accounting of messages that cannot be handed to the MQTT client at the source.

If neither condition of H2 is met within the tested range, the highest tested load is reported as a lower bound of the sustainable throughput. If no loss occurs, H3 cannot be tested. In addition, the load generator must not limit the system under test. This validity condition is checked by comparing the number of delivered records with the scheduled number per stage.

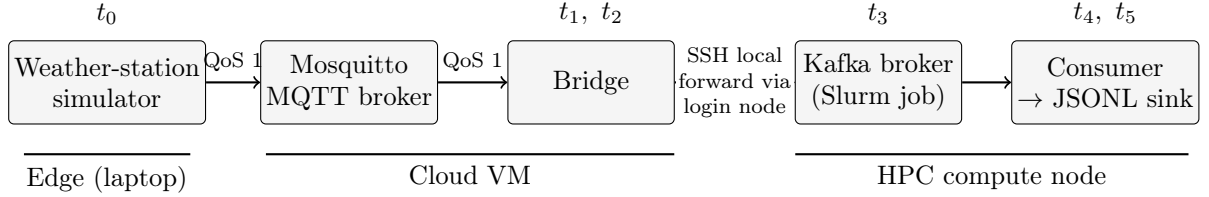
4.3 Setup and Measurement Design

Figure 1 shows the system under test and the measurement points. Simulated weather stations publish readings via MQTT to a Mosquitto broker on a cloud VM. A bridge on the same VM forwards them through an SSH local forward via the HPC login node to a Kafka broker inside a Slurm job, where a co-located consumer writes each record to a JSONL file. In the load ramp, the simulator runs on the VM instead of the laptop. MQTT uses QoS 1 for publishing and subscribing. The Kafka producer uses `acks=all` and a batching window of 5 ms (`linger.ms`), the topic has a single partition on a single

Table 4: Hypotheses for RQ2 and decision rules.

	Hypothesis	Measured by	Supported if
H1	The wide-area segment between edge and cloud dominates end-to-end latency and exceeds the segment from bridge to consumer ($t_4 - t_2$) at equal load	One-way WAN estimate ($\text{RTT}/2$) vs. median $t_4 - t_2$ at 100 msg/s; sensitivity check with $\text{RTT}/2$ plus the VM-local edge-to-bridge segment ($t_1 - t_0$) measured in the load ramp at 100 msg/s	WAN estimate $>$ median $t_4 - t_2$
H2	With increasing load, the VM-to-broker segment ($t_3 - t_2$), which contains the SSH tunnel, is the first segment whose latency grows continuously over the course of a stage, and the delivered throughput falls below the offered load	Median and p95 per segment; delivered vs. offered throughput	Both conditions hold (sustainable-throughput criterion [Kar+18])
H3	With MQTT QoS 1, message loss originates at the edge rather than on the wide-area link	Source-side drops and publish errors vs. sequence gaps per station in the sink	Every observed loss corresponds to a source-side error

broker, and the consumer commits offsets manually after flushing the sink. Each Kafka record carries approximately 200 B (estimated from a representative record). Station phases are staggered, and overdue scheduler ticks are skipped rather than caught up. Appendix A lists the test environment and run details.

Figure 1: System under test with measurement points t_0 – t_5 .

Every message carries a station identifier, a per-station sequence number and its generation time t_0 , which is stamped immediately before publishing; scheduler lateness is reported separately and not included in latency. The bridge records reception from the broker (t_1) and hand-off to the producer (t_2). The Kafka broker assigns the record timestamp t_3 on append (`LogAppendTime`), which makes t_3 independent of the producer clock [HMU20]; segments involving t_3 are evaluated only for records carrying this timestamp type. Since Kafka timestamps have millisecond resolution, t_3 is accurate to within 1 ms, and up to 1 ms may be attributed to the wrong side of the boundary between $t_3 - t_2$ and $t_4 - t_3$ [Pax98]; $t_4 - t_0$ is unaffected. The consumer records reception (t_4) and persistence (t_5). Table 5 lists the resulting segments.

Besides the tunnel, $t_3 - t_2$ includes the producer’s batching window and the broker append, so that a bottleneck in this segment cannot be attributed to the tunnel alone. End-to-end latency measured from t_0 corresponds to event-time latency [Kar+18]. Segments spanning two machines include the relative offset of their clocks [Pax98]. Since broker and consumer share a node, $t_4 - t_3$ is free of clock differences and serves as a proxy for consumer lag; offset-based lag was not recorded. Loss is detected as gaps in the sequence numbers per station, duplicates as repeated pairs of station identifier and sequence number. Throughput is computed on the consumer side as the number of received records

Table 5: Measured segments.

Segment	Name	Clocks involved
$t_1 - t_0$	Edge to bridge (WAN; VM-local in the load ramp)	Laptop and VM; VM only in the load ramp
$t_2 - t_1$	VM-internal	VM
$t_3 - t_2$	VM to broker	VM and HPC node
$t_4 - t_3$	Broker to consumer	HPC node
$t_5 - t_4$	Persistence	HPC node
$t_4 - t_0$	End-to-end; cloud-to-HPC in the load ramp	Laptop or VM, and HPC node

divided by the interval between the first and the last reception (t_4). It thus reflects the average delivery rate per stage and slightly overestimates the offered load by one record interval; transient stalls are not visible in this metric and are assessed via latency.

To assess clock synchronization, the laptop’s offset was queried via SNTP (Simple Network Time Protocol) against two NTP servers, and the synchronization state of the VM was read from its time service. The clock of the HPC node was not checked. As a clock-independent estimate of the wide-area segment, the round-trip time was measured as TCP connection-setup time from the laptop to the broker port (five samples, median), and half of it serves as one-way estimate under the assumption of a symmetric path [Pax98].

4.4 Experiments and Analysis

Table 6: Experiments.

Experiment	Purpose	Load	Environment	Hypothesis
Phase 0	Validate pipeline and instrumentation	One-factor-at-a-time sweep, 7 points $\times$ 3 repetitions, 10–1,000 msg/s	Laptop, local containers, no tunnel	–
gwdg01	Feasibility of the full chain	6 msg/s, 30 s	Laptop $\rightarrow$ VM $\rightarrow$ tunnel $\rightarrow$ HPC	–
gwdg02 with clock and RTT check	Segment breakdown, WAN estimate	100 msg/s, 60 s	Laptop $\rightarrow$ VM $\rightarrow$ tunnel $\rightarrow$ HPC	H1
Load ramp (T1)	Latency and reliability under load	100, 500, 1,000, 2,000 and 5,000 msg/s, 60 s each	Generator on VM $\rightarrow$ tunnel $\rightarrow$ HPC	H2, H3, validity condition
Fault injection (T2)	Behavior during tunnel interruption	100 msg/s, interruptions of 15 s and 45 s	VM, tunnel	H3

In Phase 0, each run used a new topic and a new consumer group. In the load ramp, stages were separated by 15 s pauses to drain in-flight messages. Bridge, tunnel, broker and consumer ran continuously after a preceding smoke test; only the simulator was restarted per stage. Stages were distinguished by station prefix within a single topic.

For each run or stage, the analysis script computes the segments per record and reports median, p95 and maximum. No warm-up period is excluded, so connection setup at the start of each load-ramp stage is included in the statistics. Phase 0 comprises three repetitions per point; the runs on GWDG infrastructure are single runs, as the project’s core-hour allocation was exhausted after the measurement campaign and HPC

access ended subsequently, and are therefore exploratory. The repository, including scripts and configuration, is made accessible (tag `report-v1`); the raw sink data remain on the GWDG project storage and are no longer accessible.

4.5 Deviations and Assumptions

Within the project’s time frame and resource access, priority was given to obtaining a working end-to-end prototype. Table 7 lists the resulting deviations from the initial project plan; except for H4, all were decided before the affected measurements.

Table 7: Deviations from the project plan.

Planned	Implemented	Reason
MQTT with TLS and authentication	Plain TCP on port 1883 without authentication; access restricted to the laptop’s IP address by a security group	Priority on the end-to-end prototype
H2 tested by a frequency sweep through the tunnel	H2 tested by the load ramp	Integrated load test on real infrastructure
H5: edge device as limiting factor	Generator moved to the VM; validity condition (Sec. 4.2)	Exclude the edge device as limiting factor
H4: comparison with file-based transfer (rsync/scp)	Not carried out	Time; end of HPC access

The design rests on three assumptions: clock offsets remain constant over the short runs; the wide-area path is symmetric for the RTT-based estimate; and the load generator, which in the load ramp shares the VM with broker and bridge, does not distort the measured segments, although this departs from the separation of load driver and system under test recommended by [Kar+18]. In addition, the load ramp excludes the wide-area segment. Further limitations and their impact on the results are discussed in Section 6.3.

5 Results

This Section presents the results for RQ1, i.e., the architecture assessed against the requirements in Table 3 (Sections 5.1 and 5.2), and for RQ2, i.e., the measurements evaluated against the decision rules in Table 4 (Sections 5.3 and 5.6). Section 5.7 answers both research questions. Interpretation in the context of related work and the full discussion of limitations follow in Section 6.

5.1 Architecture and Requirement Fulfilment

Three options for connecting the cloud VM to a broker on a compute node were considered (Table 8). They differ in which side initiates the connection and thus in the facility-side prerequisites. Option C would require an outbound connection from inside a job and a private SSH key on the cluster, which is not permitted [GWD26b]; it was therefore

excluded. Option B places Kafka permanently on the VM and lets the job consume outward, a pull pattern comparable to RBF [Kur+26]. It decouples the broker from the job time limit, but requires outbound connectivity of compute nodes, which is limited at GWDG [GWD26b]. Option A was implemented as the only option that requires no administrative change. In terms of the taxonomy of George et al. [Geo+25], it instantiates PRS at user level: the user-operated VM acts as proxy and the login node as the known gateway.

Table 8: Options for connecting the cloud VM to the HPC broker.

	A: Forward tunnel from VM, Kafka in job (implemented)	B: Kafka on VM, job consumes outward	C: Reverse tunnel from job
Connection initiated by	VM $\rightarrow$ login node $\rightarrow$ compute node	Job $\rightarrow$ VM	Job $\rightarrow$ VM
Facility-side prerequisite	None beyond a regular account	Outbound connectivity of compute nodes	Outbound connectivity and a private key on the cluster
R1 (regular user rights)	technically met; permanent tunnel an open policy question	without exception	not permitted
R7 (job time limit)	broker ends with the job	broker outside the job	broker ends with the job
Implementation effort	Moderate: node discovery, listener configuration	Low once permitted	Moderate
Predictable failure risk	Single connection via shared login node, tied to a user session	Not assessed; depends on the outbound link from the job	Not assessed (excluded)

Table 9 assesses the implemented architecture: two requirements are met, four partially and one not at all. R1 is met technically, as the pipeline was built and operated without any administrative change, but only partially in a stricter sense. The operators of SAIA at the same facility regard port forwarding to compute nodes and SSH keys on external servers as risks [Doo+26], and the prototype avoided a key on the VM only by forwarding the SSH agent from the user’s machine, which ties the tunnel to an active user session. Whether such a tunnel may be operated permanently thus remains an open policy question. R7 is not met: since the broker runs inside the job and stores its log in a job-specific directory in the node’s temporary storage, records not consumed before the job ends are lost, and consumption cannot resume in a later job. These two requirements mark the main trade-off against Option B, which would meet R7 but presupposes an exception to the network policy.

5.2 Implementation Findings

Table 14 in Appendix A lists the obstacles that required the most effort. The most consequential ones concerned the tunnel: the broker had to advertise the client side of the tunnel, and the job had to publish its node at runtime. Most obstacles concerned

Table 9: Fulfilment of requirements R1–R7 (✓ met, ● partially met, × not met).

ID	Status	Implementation and evidence
R1	●	No administrative change needed; a permanent server-held tunnel is assessed as a risk by the facility’s operators; agent forwarding requires an active user session
R2	✓	Job publishes its node name on the shared file system, the tunnel script reads it via the login node; Kafka advertises <code>localhost</code> so that clients reconnect through the tunnel
R3	●	MQTT with QoS 1 and automatic reconnection; no TLS or authentication; behavior under link loss not tested
R4	●	Kafka decouples bridge and consumer; the bridge buffers only transiently in memory, and its failure leads to silent loss (Section 5.6)
R5	✓	Per-station sequence numbers detect intermediate loss and duplicates; complete or tail loss is detected only by comparison with the scheduled count (Sections 5.5 and 5.6)
R6	●	Single-clock segments $t_2 - t_1$, $t_4 - t_3$ (via <code>LogAppendTime</code>) and $t_5 - t_4$; the wide-area segment was not directly measurable, and segments spanning VM and HPC node include an unchecked clock offset (Section 5.4)
R7	×	Broker storage ends with the job; manual offset commits would allow consumer restarts within a job (not tested)

network access, configuration across tiers and platform policies; the others concerned the runtime environment and the consumer’s buffering logic. None concerned the generation or processing of the data themselves.

5.3 Functional Validation

In Phase 0, all 21 local runs delivered every message without loss at rates up to 1,000 msg/s, which validated data flow and instrumentation before the integration on GWDG infrastructure. In `gwdg01`, the first run across the full chain, all 180 records arrived with all six timestamps, and t_3 carried `LogAppendTime` in every record, confirming the broker-side timestamping on which the single-clock segments depend.

5.4 Latency Breakdown on Real Infrastructure

Figure 2 shows the segment medians at 100 msg/s for `gwdg02`, with the laptop as source, and for the first load-ramp stage (`r0100`), with the generator on the VM.

The bridge and the sink contribute almost nothing: the VM-internal segment stays below 0.05 ms in both runs, and persistence takes 0.014 ms in `gwdg02`. Behind the bridge, the path is dominated by the VM-to-broker segment of about 6.5 ms (6.51 ms in `gwdg02`, 6.48 ms in `r0100`), which contains the tunnel and the producer’s 5 ms batching window (`linger.ms`). At 100 msg/s, i.e., one record every 10 ms on average, each record typically opens a new batch and waits for the full window, which leaves roughly 1.5 ms for the tunnel and the broker append. The broker-to-consumer segment adds 1.40 ms and 1.05 ms, respectively, and the median of the combined segment $t_4 - t_2$ is 7.84 ms in `gwdg02` (p95 9.08 ms). Since $t_3 - t_2$ and $t_4 - t_2$ span the VM and HPC clocks and the HPC clock was not checked, these values include an unknown relative offset. In addition,

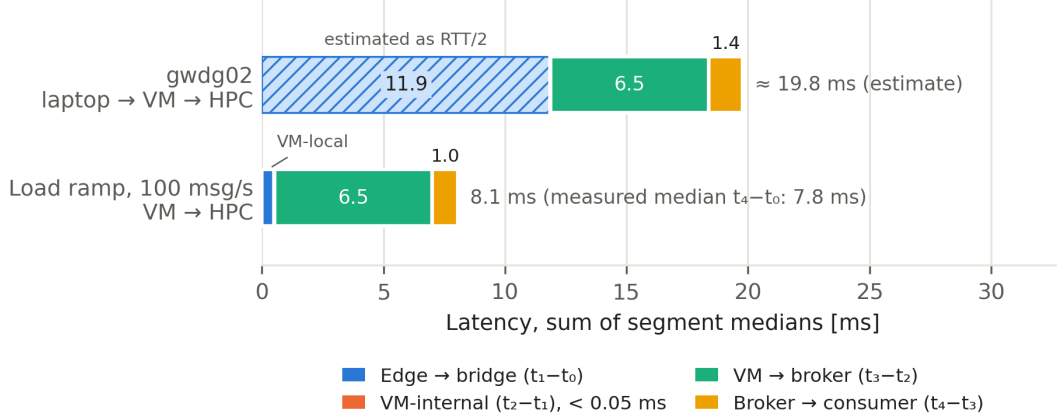


Figure 2: Segment medians at 100 msg/s on GWDG infrastructure. For **gwdg02**, the edge-to-bridge segment is estimated as half the round-trip time because of the laptop’s clock offset (hatched); in **r0100**, the generator ran on the VM. Since medians are not additive, stacked bars differ slightly from the median of a combined segment (e.g., 7.91 vs. 7.84 ms for $t_4 - t_2$ in **gwdg02**). The VM-internal segment is below 0.05 ms in both runs.

Kafka truncates timestamps to milliseconds, so that $t_3 - t_2$ is underestimated and $t_4 - t_3$ overestimated by up to 1 ms, whereas $t_4 - t_2$ is unaffected (Section 4.3). The agreement of both runs, executed in different Slurm jobs, within 0.03 ms for the VM-to-broker segment and within 0.4 ms for the broker-to-consumer segment suggests that the offset is stable. If each record waits for the full batching window, the VM-to-broker median of 6.48 ms limits a lead of the HPC clock over the VM clock to about 1.5 ms; a lag cannot be bounded in this way, so the offset cannot be quantified.

The measured edge-to-bridge segment of **gwdg02** (median 236.6 ms) cannot be interpreted. SNTP queries against two servers, 13 minutes apart, showed the laptop’s clock 0.21 ± 0.02 s and 0.23 ± 0.02 s behind the reference, whereas the VM was synchronized to a GWDG stratum-1 server with an offset close to zero. Subtracting these offsets from the measured medians of **gwdg01** and **gwdg02** yields -15 ± 16 ms and $+6 \pm 23$ ms, values that are consistent with the round-trip estimate but too imprecise to replace it. The round-trip time between laptop and broker port was 23.7 ms (median of five samples), giving a one-way estimate of 11.9 ms. Together with the segments behind the bridge, the estimated end-to-end latency is about 20 ms, of which the wide-area segment accounts for about 60 %.

H1 is supported: the wide-area estimate (11.9 ms) exceeds the median bridge-to-consumer segment (7.84 ms), and adding the VM-local edge-to-bridge segment from **r0100** (0.50 ms) only widens the gap. The support rests on an estimate rather than a direct one-way measurement, and the comparison includes the unknown offset between the VM and HPC clocks.

5.5 Behavior under Load

In every stage of the load ramp, the number of delivered records matched the scheduled number exactly (Table 10). The validity condition is therefore met: the generator on the VM did not limit the system under test.

Table 10: Load ramp: load, data rate, loss and per-segment latency (cloud-to-HPC path, source on cloud VM).

Offered load [msg/s]	100	500	1,000	2,000	5,000
<i>Volume and reliability</i>					
Data rate ¹ [MB/s]	≈ 0.02	≈ 0.10	≈ 0.20	≈ 0.40	≈ 1.00
Throughput [msg/s]	100.05	500.05	1,000.05	2,000.12	5,000.33
Messages	6,000	30,000	60,000	120,000	300,000
Lost / duplicates	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
<i>Latency [ms]</i>					
VM → broker ($t_3 - t_2$) – median	6.48	4.63	4.31	5.20	5.71
VM → broker ($t_3 - t_2$) – p95	7.74	7.48	7.28	2,016.70 ²	8.48
Broker → consumer ($t_4 - t_3$) – median	1.05	0.88	1.06	1.29	1.72
Broker → consumer ($t_4 - t_3$) – p95	1.55	1.44	2.00	15.25	3.02
Cloud-to-HPC ($t_4 - t_0$) – median	7.80	5.83	5.87	6.89	8.65
Cloud-to-HPC ($t_4 - t_0$) – p95	9.13	8.99	9.71	2,032.64 ²	14.80

¹ Approximate data rate into the tunnel, estimated from a representative record (~200 B Kafka record value incl. bridge timestamps). Protocol, batching and SSH overhead are excluded.

² Single transient stall of ~4.1 s on the VM → broker segment; 8,174 messages were delayed, all were delivered. Cause not determined.

Across all stages, 516,000 records were delivered without loss or duplicates, and the average delivery rate matched the offered load. The median cloud-to-HPC latency follows a U-shape (Figure 3): it falls from 7.80 ms at 100 msg/s to 5.83 ms and 5.87 ms at 500 and 1,000 msg/s and rises again to 8.65 ms at 5,000 msg/s. The decrease originates in the VM-to-broker segment (6.48 to 4.31 ms) and is consistent with the batching window. At 100 msg/s, each record typically opens a new batch and waits for the full 5 ms, whereas at higher rates more records join an already open batch and wait only for the remainder of the window. Batches are not filled, since even at 5,000 msg/s only about 5 KB accumulate per window, well below the default batch size of 16,384 B [HMU20]. Beyond 1,000 msg/s, the medians of the VM-to-broker segment (4.31 to 5.71 ms) and the broker-to-consumer segment (1.06 to 1.72 ms) rise moderately; the cause was not isolated. The p95 stays below 10 ms up to 1,000 msg/s and reaches 14.80 ms at 5,000 msg/s, and the VM-internal segment remains below 0.07 ms at all loads. At 5,000 msg/s, the p95 of the VM-local edge-to-bridge segment rises from at most 1.26 ms in the lower stages to 3.95 ms. Since the generator shares the VM with Mosquitto and the bridge, this increase cannot be separated from the load of the generator itself (Section 4.5).

The stage at 2,000 msg/s deviates. Its median is unremarkable, but the p95 of the VM-to-broker segment reaches 2.0 s. The cause is a single stall between 21:39:44 and 21:39:48 UTC: 8,174 records produced within 4.1 s were delayed by more than 1 s, at most by 4.8 s, and were all delivered afterwards. As these records make up 6.8 % of the stage, this single event fully accounts for the p95. Whether the shared login node, the network or the broker caused the stall could not be determined; as it did not recur at 5,000 msg/s, it is not indicative of a capacity limit. In addition, individual consumer outliers of about 250–390 ms occurred in four of the five stages without affecting the p95 of that segment.

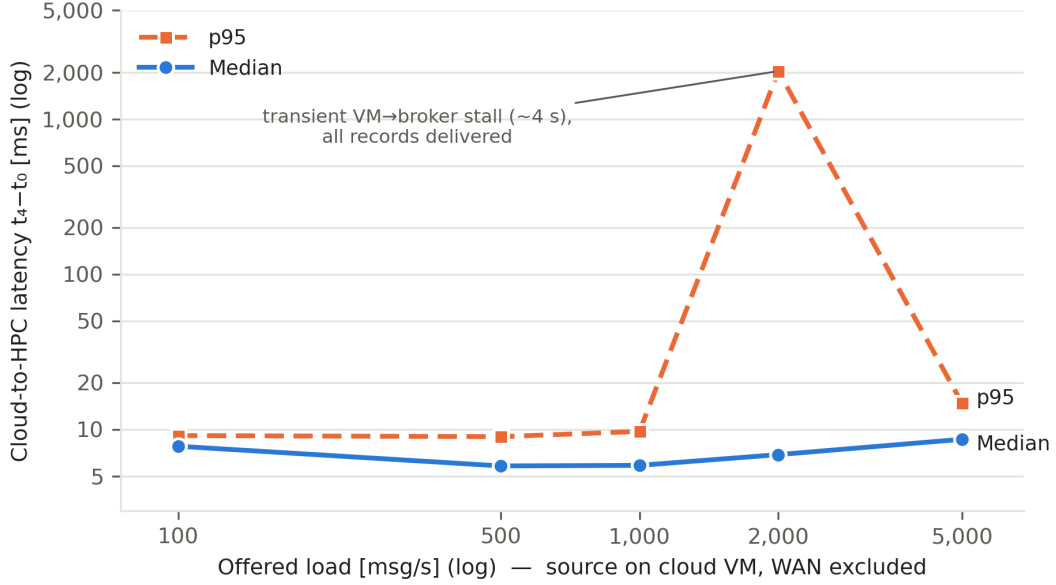


Figure 3: Cloud-to-HPC latency (median and p95) versus offered load. Both axes are logarithmic. The p95 peak at 2,000 msg/s reflects a single transient stall on the VM → broker segment, during which all delayed records were delivered without loss.

H2 is not supported within the tested range, as neither condition of the decision rule holds. The delivered throughput matched the offered load in every stage, and no segment showed a continuously growing latency. Such growth over the course of a stage would separate p95 and median widely, whereas the cloud-to-HPC p95 exceeds the median by at most 6.2 ms in all stages except the one with the transient stall. This spread widens with load, from 1.3 ms at 100 msg/s to 6.2 ms at 5,000 msg/s, which is consistent with the first signs of pressure at 5,000 msg/s noted above; a time-resolved analysis that could separate these from beginning backpressure is not possible, as the raw data are no longer accessible. The moderate increase of some segment medians is thus consistent with load dependence rather than sustained backpressure. Following the decision rule in Section 4.2, 5,000 msg/s, or about 1 MB/s, is reported as a lower bound of the sustainable throughput. This bound applies to small records and to stages of 60 s.

5.6 Reliability and Fault Behavior

No record was lost or duplicated in any valid run, neither in Phase 0 nor on GWDG infrastructure, where `gwdg01`, `gwdg02` and the load ramp comprise 522,180 records. Since no loss occurred, H3 cannot be tested according to the decision rule in Section 4.2.

The fault injection did not yield a valid result. The tunnel was interrupted as planned for 15 s and 45 s, but post-hoc inspection showed that the bridge had already terminated before the first run, so the precondition of an intact pipeline was violated. A repetition was not possible because the project’s core-hour allocation was exhausted. The runs nevertheless yielded an observation relevant to reliability. The simulator registered every publish as successful, since it counts a message as sent once the MQTT client has accepted it. Even an acknowledgement by the broker would only have confirmed receipt by Mosquitto, as QoS 1 covers a single client–broker hop (Section 2.2). With the bridge down, Mosquitto had no active subscriber, and since the bridge connected with a non-persistent session,

the default of paho-mqtt, the messages were not retained [BG14]. The loss was thus silent for the source and became visible only when comparing published and received counts, which illustrates the end-to-end argument introduced in Section 2.2 [SRC84]. The only evidence of robustness under disturbance is therefore the transient stall at 2,000 msg/s, during which all delayed records were delivered without loss.

5.7 Answers to the Research Questions

Table 11 summarizes the results with respect to the research questions and hypotheses.

Table 11: Results with respect to research questions and hypotheses.

	Result	Status	Limits
RQ1	A complete pipeline can be built with open-source tools and regular user rights as a user-level PRS	Answered with qualification	Permanent tunnel operation an open policy question (R1); broker state ends with the job (R7)
RQ2	Cloud-to-HPC median 5.83–8.65 ms up to 5,000 msg/s; $p_{95} \leq 15$ ms except one transient stall; full chain ≈ 20 ms (estimated); no loss in fault-free operation	Partially answered	Wide-area segment estimated; small records; reliability under faults not shown
H1	Wide-area estimate 11.9 ms > bridge to consumer 7.84 ms; ≈ 60 % of estimated end-to-end latency	Supported	RTT/2 estimate; HPC clock not checked
H2	No sustained backpressure; throughput equals offered load up to 5,000 msg/s (≈ 1 MB/s)	Not supported in tested range	Small records; single 60 s stages
H3	No loss occurred	Not testable	Fault injection invalid

The research questions are answered to different degrees. RQ1 is answered with qualification: the prototype shows that a complete edge-to-HPC streaming pipeline can be built with open-source tools under real data-center conditions and without administrative changes. Two requirements remain partially or not met: the permanent operation of a user-held tunnel is an open policy question (R1), and the broker state does not survive the end of a job (R7). RQ2 is answered in part. For the path from the cloud VM to the HPC consumer, latency and throughput under varying load are well characterized for small records. For the full chain, the wide-area segment could only be estimated, and reliability is demonstrated only for fault-free operation, so that H3 remains untested.

The results are subject to four main limitations: the clock offset of the edge device, which prevents a direct measurement of the wide-area segment; the unchecked clock of the HPC node; single runs on a shared compute node, with the load generator sharing the VM with Mosquitto and the bridge; and the invalid fault injection. In addition, the raw sink data are no longer accessible, so the results cannot be re-analyzed at record level. Section 6 discusses their impact.

Three implications follow for the research questions. First, the user-level tunnel is not the limiting factor at the tested scale: up to 5,000 msg/s, the cloud-to-HPC path adds about 6–9 ms in the median, of which the producer’s batching window accounts for a large share at low load, so that latency behind the bridge is governed by configuration as much as by the tunnel. Second, according to the estimate, the wide-area segment dominates the end-to-end latency of the full chain. Third, the main constraints on RQ1 are operational rather than performance-related: permanent operation depends on policy, data do not survive the job, and a failing bridge loses data silently. The results thus indicate that the bottleneck of such a pipeline lies in establishing and operating connectivity across the seams between tiers rather than in transport overhead or data processing. Section 6 places this finding in the context of related work.

6 Discussion

This section interprets the results in the context of related work, derives practical implications and discusses the limitations of the study and the questions they leave open.

6.1 Interpretation

The results indicate that a user-level edge-to-HPC pipeline is limited by connectivity and policy rather than by performance. George et al. [Geo+25] made a similar observation for facility-operated infrastructure, where network policy rather than performance hindered the external use of PRS [Geo+25]. This work extends the observation to the perspective of a regular user: PRS can be instantiated at user level with standard tools, but its permanent operation falls into the same policy domain, as the facility’s own operators assess such tunnels as a risk [Doo+26]. Technical feasibility, which the experiments demonstrate, and operational permissibility, which they cannot establish, thus diverge.

Behind the bridge, latency is governed less by the tunnel than by producer batching, which trades latency for throughput [HMU20]. At low rates, the 5 ms batching window dominates this part of the path, which explains the U-shaped median under increasing load. This latency is therefore largely a matter of configuration, whereas the wide-area segment determines the end-to-end latency (H1). As the lower bound of about 1 MB/s lies far below the throughput reported for a single Kafka partition [HMU20], the capacity of the user-level path remains open.

The silent loss after the bridge had failed illustrates the end-to-end argument: acknowledgements on individual hops do not add up to end-to-end delivery [SRC84]. Related work closes such gaps through persistence, by buffering data at the edge [Jav+18] or by exchanging them via logs placed outside the firewalls [Kur+26]. A persistent buffer in the bridge could be added at user level, but retaining data beyond the end of a job requires a broker outside the job, i.e., Option B and thus a policy exception. Robustness and user-level operation are therefore in a trade-off that users cannot resolve on their own.

6.2 Practical Implications

For users, the pattern is reproducible with regular rights and suits time-bounded, supervised campaigns within a job, such as near-real-time monitoring at low data rates, but not unattended continuous operation or applications that must exclude data loss. Its

most transferable lesson is that an end-to-end check, e.g., sequence numbers per source, is indispensable, since broker acknowledgements do not reveal loss behind the bridge. For operators, a robust variant requires a sanctioned option: a restricted persistent tunnel endpoint, outbound connectivity of compute nodes, or a facility-operated streaming endpoint in the sense of MSS [Geo+25].

6.3 Limitations and Open Questions

Three limitations restrict the validity of the results. First, the clock of the edge device deviated by about 0.2 s, although NTP can keep clocks within a few milliseconds of reference time [Mil91], and the clock of the HPC node was not checked. H1 therefore rests on an RTT-based estimate that assumes a symmetric path [Pax98]; a direct one-way measurement would require disciplined synchronization on all three tiers. Second, the load ramp consisted of single runs on a shared node, with small records, without the wide-area segment and with the load generator co-located with broker and bridge, contrary to the separation of load driver and system under test recommended by [Kar+18]. Its results are therefore exploratory; locating the capacity limit requires repetitions with a separate load driver, larger records and a time-resolved analysis. Third, the invalid fault injection leaves H3 and the behavior under link failures untested; a repetition should be combined with a persistent buffer in the bridge.

The greatest implementation effort concerned reachability, i.e., the advertised address through the tunnel and node discovery at runtime, rather than data processing (Table 14), which itself supports the central finding. The end of HPC access after the measurement campaign prevented repetitions and a valid fault injection. Finally, the prototype omitted TLS (Transport Layer Security) and authentication and is therefore not suited for productive use.

7 Conclusion

This report investigated how continuous edge data streams can be connected to HPC resources with the rights of a regular user. Regarding RQ1, a three-tier prototype at GWDG demonstrated that a complete edge-to-HPC streaming pipeline can be built with open-source tools under real data-center conditions and without administrative changes: MQTT carries the data from the edge to a cloud VM, and an SSH local forward via the login node connects a bridge to Kafka inside a Slurm job. This answer is qualified, as the permanent operation of such a tunnel remains an open policy question and the broker state does not survive the end of a job. Regarding RQ2, instrumented experiments showed that the path from cloud to HPC adds a median latency of about 6–9 ms at loads up to 5,000 msg/s without loss in fault-free operation, and that the wide-area segment dominates the estimated end-to-end latency of about 20 ms. Reliability under faults could not be demonstrated.

The central finding is that the bottleneck of such a pipeline lies not in transport or data processing, but in establishing and operating connectivity across the seams between tiers. The report thus contributes a user-level instantiation of proxied streaming, a segment-wise measurement design that separates single-clock from cross-clock segments, and an empirical characterization of this path on real infrastructure.

Future work should measure the wide-area segment directly with disciplined clock synchronization, locate the capacity limit with repeated runs and larger records, and test robustness under faults with a persistent buffer in the bridge. Beyond that, a comparison with file-based transfer and a sanctioned architecture with persistence outside the job, developed together with the facility, would show how far the approach can be carried into productive use.

References

- [BG14] *MQTT Version 3.1.1*. OASIS Standard. OASIS, Oct. 2014. URL: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [Chu+22] Joaquin Chung et al. “SciStream: Architecture and Toolkit for Data Streaming between Federated Science Instruments”. In: *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC ’22)*. Minneapolis, MN, USA: ACM, 2022, pp. 185–198. DOI: 10.1145/3502181.3531475.
- [Doo+26] Ali Doosthosseini et al. “SAIA: A Seamless Slurm-Native Solution for HPC-Based Services”. In: *The Journal of Supercomputing* 82 (2026). Article 403. DOI: 10.1007/s11227-026-08508-3.
- [Geo+25] Anjus George et al. “From Edge to HPC: Investigating Cross-Facility Data Streaming Architectures”. In: *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops ’25)*. St. Louis, MO: ACM, Nov. 2025, pp. 949–959. DOI: 10.1145/3731599.3767452.
- [GWD26a] GWDG. *Apache Spark*. Documentation for HPC. Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen. 2026. URL: https://docs.hpc.gwdg.de/software_stacks/parallelization/apache_spark/index.html (visited on 07/04/2026).
- [GWD26b] GWDG. *Data Transfers*. Documentation for HPC. Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen. 2026. URL: https://docs.hpc.gwdg.de/how_to_use/data_transfer/index.html (visited on 07/04/2026).
- [HMu20] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. “How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka”. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2020, pp. 641–648. DOI: 10.1109/ICPADS51040.2020.00089.
- [Jav+18] Asad Javed et al. “CEFIoT: A Fault-Tolerant IoT Architecture for Edge and Cloud”. In: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. Singapore: IEEE, 2018, pp. 813–818. DOI: 10.1109/WF-IoT.2018.8355149.
- [Kar+18] Jeyhun Karimov et al. “Benchmarking Distributed Stream Data Processing Systems”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. Paris: IEEE, 2018. URL: <https://ieeexplore.ieee.org/document/8509390>.
- [KNR11] Jay Kreps, Neha Narkhede, and Jun Rao. “Kafka: A Distributed Messaging System for Log Processing”. In: *Proceedings of the International Workshop on Networking Meets Databases (NetDB ’11)*. Athens, June 2011, pp. 1–7.
- [Kur+26] Liubov Kurafeeva et al. *Hybrid Edge-HPC Systems for Low-Latency Data-Driven Inference*. Version 2. Preprint. 2026. arXiv: 2605.20532 [cs.DC].

- [LCJ18] Andre Luckow, George Chantzialexiou, and Shantenu Jha. “Pilot-Streaming: A Stream Processing Framework for High-Performance Computing”. In: *2018 IEEE 14th International Conference on e-Science (e-Science)*. Amsterdam, Netherlands: IEEE, 2018, pp. 177–188. DOI: 10.1109/eScience.2018.00033.
- [LRJ21] Andre Luckow, Kartik Rattan, and Shantenu Jha. “Pilot-Edge: Distributed Resource Management Along the Edge-to-Cloud Continuum”. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 874–878. DOI: 10.1109/IPDPSW52791.2021.00130.
- [Mil91] David L. Mills. “Internet Time Synchronization: The Network Time Protocol”. In: *IEEE Transactions on Communications* 39.10 (Oct. 1991), pp. 1482–1493.
- [Nai17] Nitin Naik. “Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP”. In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. IEEE, Oct. 2017. DOI: 10.1109/SysEng.2017.8088251.
- [Pax98] Vern Paxson. “On Calibrating Measurements of Packet Transit Times”. In: *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98)*. Madison, WI: ACM, June 1998, pp. 11–21.
- [SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. “End-to-End Arguments in System Design”. In: *ACM Transactions on Computer Systems* 2.4 (Nov. 1984), pp. 277–288. DOI: 10.1145/357401.357402.
- [Wan+21] Guozhang Wang et al. “Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka”. In: *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Virtual Event, China: ACM, June 2021, pp. 2602–2613. DOI: 10.1145/3448016.3457556.
- [YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In: *Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*. Vol. 2862. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3.
- [YL06] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. IETF, Jan. 2006. DOI: 10.17487/RFC4254.

A Test Environment and Run Details

Table 12: Test environment.

Component	Version / configuration	Tier
Edge device	MacBook Air M2, 8 GB RAM; Python 3.11	Laptop
Cloud VM	GWDG OpenStack m1.xlarge: 8 vCPUs, 32 GB RAM; Ubuntu 24.04; Python 3.12	Cloud
Mosquitto	2.0.18; listener on port 1883, anonymous access	Cloud VM
paho-mqtt	2.1.0	Laptop (simulator), cloud VM (bridge; simulator in load ramp)
kafka-python-ng	2.2.3	Cloud VM (bridge), HPC (consumer)
Apache Kafka	3.9.1 (Scala 2.13), KRaft mode, single broker; <code>log.message.timestamp.type=LogAppendTime</code>	HPC compute node
Slurm allocation	Partition <code>scc-cpu</code> ; 1 node, 1 task, 4 cores, 1 h walltime	HPC
Compute node	2× Intel Xeon Platinum 8468 (Sapphire Rapids), 96 cores, shared	HPC
Java / Python	OpenJDK 17.0.11; Python 3.11.9	HPC
Pipeline configuration	MQTT QoS 1; producer <code>acks=all</code> , <code>retries=3</code> , <code>linger.ms=5</code> , no compression, key <code>station_id</code> ; topic auto-created with broker defaults, 1 partition, replication factor 1; consumer with manual commits after 100 records or 1 s	All tiers

Local validation (Phase 0) used the container images `apache/kafka:3.9.1` and `eclipse-mosquitto:2` (major-version tag).

Table 13: Run details.

Run	Stations $\times$ frequency	Offered load	Duration	Records	Remarks
Phase 0	Baseline 10×10 Hz, payload 0 B; factors: stations {1, 10, 50}, frequency {1, 10, 100} Hz, padding {0, 512, 4,096} B	10–1,000 msg/s	60 s per run		– 3 repetitions per point; per run new topic and consumer group, 10 s wait for group join, 5 s cool-down
gwdg01	3×2 Hz	6 msg/s	30 s	180	30 Sep 2026; Slurm job 16377586, node c0409; prefix station
gwdg02	10×10 Hz	100 msg/s	60 s	6,000	Same job and node; prefix gw02
Smoke test	5×10 Hz	50 msg/s	10 s	500	Before the load ramp; created the topic
r0100	10×10 Hz	100 msg/s	60 s	6,000	Load ramp, 30 Sep 2026
r0500	10×50 Hz	500 msg/s	60 s	30,000	Load ramp
r1000	20×50 Hz	1,000 msg/s	60 s	60,000	Load ramp
r2000	20×100 Hz	2,000 msg/s	60 s	120,000	Load ramp
r5000	50×100 Hz	5,000 msg/s	60 s	300,000	Load ramp
a15 / a45	10×10 Hz	100 msg/s	120 s each		– Fault injection: tunnel interrupted after 30 s for 15 s / 45 s; runs invalid, see Section 5.6

All runs without payload padding (except the Phase-0 padding factor), with staggered station phases and skipped overdue ticks; 15 s pauses between load-ramp stages.

Table 14: Selected implementation obstacles.

Obstacle	Cause	Solution	Transferable insight
Kafka clients could not reconnect through the tunnel	Brokers return their advertised address, which must be reachable from the client	Advertise <code>localhost</code> , bind to all interfaces	In tunneled setups, the advertised address must refer to the client side of the tunnel
Target node unknown before the job starts	Allocation by the scheduler	Job writes its node name to the shared file system; port check against a second job on the same node	Dynamic discovery and isolation must be part of the job script
Run identifiers did not reach the job	Variables set before <code>sbatch</code> were not propagated	Explicit <code>-export</code>	Job configuration must be passed explicitly
Consumer latency in the range of seconds	Debug logging written to the shared file system	Reduced log level	Instrumentation itself can dominate latency on shared storage
Records remained buffered at low load	Sink flush and offset commit only after a fixed number of records	Additional time-based flush and commit	Count-based batching needs a time bound in low-rate streams
Bridge library failed on the VM	<code>kafka-python</code> incompatible with Python 3.12	Maintained fork <code>kafka-python-ng</code>	Client libraries must be checked against each tier's runtime
VM access failed initially	Cloud platform rejected Ed25519 keys; security group admitted only registered source addresses	RSA key; rule updated for each new source address	Cloud-side access policies shape the setup as much as HPC policies

B Code

The project repository can be found at the following link: <https://gitlab.gwdg.de/jan.goewecke01/edge-hpc-pipeline.git>

C Transparency Report: Tools and Third-Party Contributions

C.1 Overview

- Writing
 - LaTeX, edited in Overleaf using the chair’s report template
- Images
 - draw.io for graphics
 - TikZ, generated with Claude Sonnet 5.5
- Diagrams
 - Python Matplotlib for the plots
- Code
 - IDE: Microsoft Visual Studio Code
 - Code generation and debugging with Claude Opus 5.5
 - LaTeX code generated and checked with Claude Sonnet 5.5
- AI
 - Claude Opus 5.5 (Anthropic): idea generation, proofreading, revising, improving, translating and shortening of texts in all chapters; generation of code and debugging
 - Claude Sonnet 5.5 (Anthropic): generation and checking of LaTeX code
- Third-Party
 - Sepehr Mahmoodian helped with the structure of the report and with finding ideas, gave regular feedback, and commented on the content of parts of the report
 - The experiments were run on the infrastructure of the GWDG (cloud VM and HPC cluster)

C.2 Per Section

- Abstract
 - Claude Opus 5.5 was used for revising, improving and shortening the text
- Section 1 Introduction
 - Claude Opus 5.5 was used for idea generation, proofreading, revising, improving, translating and shortening
 - Sepehr Mahmoodian provided feedback on the structure and ideas
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter
- Section 2 Background
 - Claude Opus 5.5 was used for idea generation, proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter, including Table 1
- Section 3 Related Work
 - Claude Opus 5.5 was used for idea generation, proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter, including Table 2, and the formatting of the BibTeX entries
- Section 4 Methodology
 - Claude Opus 5.5 was used for idea generation, proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter, including the tables and the TikZ figure of the system architecture (Figure 1)
 - Code of the prototype and the measurement setup: generation and debugging with Claude Opus 5.5
- Section 5 Results
 - Claude Opus 5.5 was used for proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter, including the tables
 - Plots created with Python Matplotlib
- Section 6 Discussion
 - Claude Opus 5.5 was used for idea generation, proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter
- Section 7 Conclusion

- Claude Opus 5.5 was used for proofreading, revising, improving, translating and shortening
 - Claude Sonnet 5.5 generated the LaTeX code of the chapter
- Appendix
 - Claude Sonnet 5.5 generated the LaTeX code of the tables in the appendix