# Monitor, Mitigate, Moderate: Backpressure in Stream Benchmark Generators

14th International Workshop on Load Testing and Benchmarking of Software Systems, **LTB 2026**
Florence, Italy

Iain Dixon, Matthew Forshaw, Joe Matthews

4th of May 2026

Newcastle University - School of Computing

# Brief Explanation of Stream Processing Systems (SPS)

# Pipelined operators process stream data

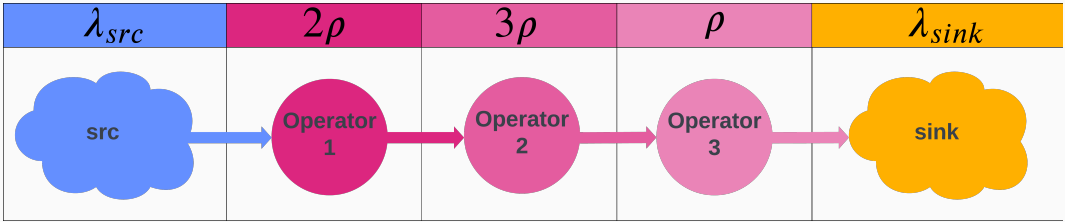



**Figure 1.** Logical layout of a Stream Processing System (SPS), with an upstream **source**, downstream **sink**, and pipelined operators. Each operator has a processing speed $x\rho$ where $x$ is the multiple of the comparitive processing speed of the minimum operator's processing speed $\rho$.

# Scaling mechanisms provision operators to meet SPS demand

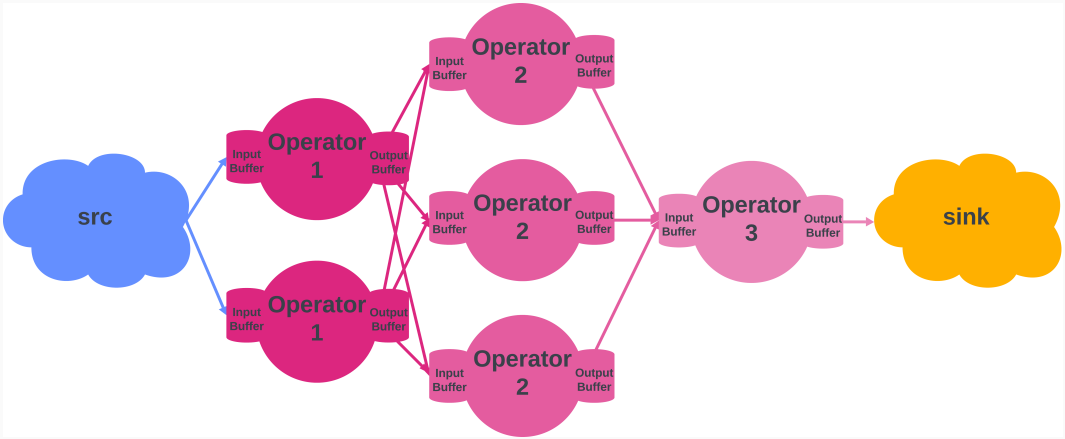



**Figure 2.** Physical layout of an SPS where the operators are scaled to handle processing loads between operators.

**DS2 Autoscaler:** Kalavri et al., "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows"

# Load shedding drops records to throttle external rates

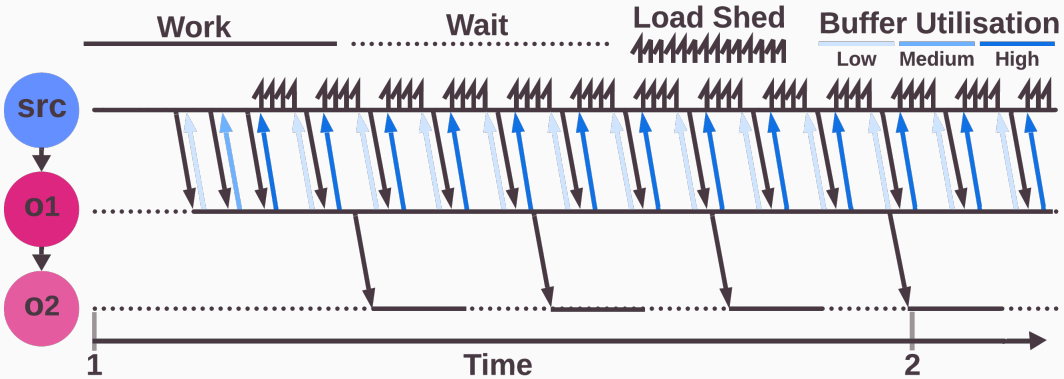



**Figure 3.** Load shedding maintains the **external** SPS rate by dropping external records received.

# Backpressure halts upstream operators to throttle internal rates

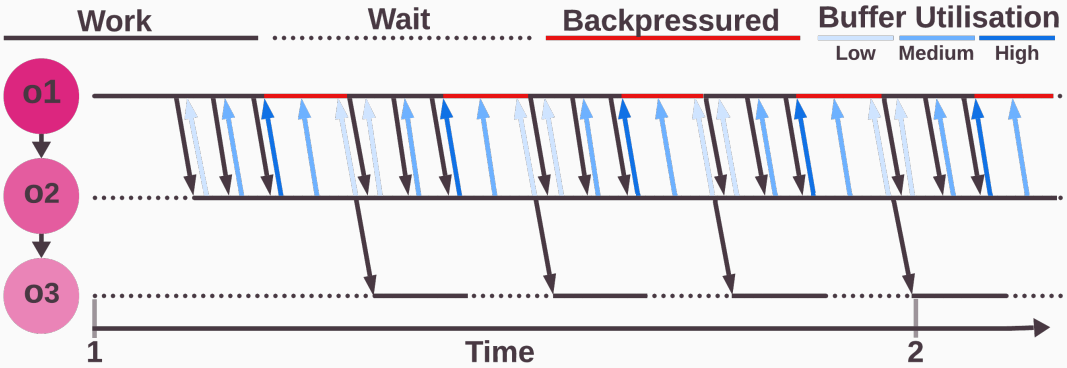



**Figure 4.** Backpressure maintains the **internal** SPS rate by slowing down internal records received.

# Open-loop SPS benchmarks require source/sink separation

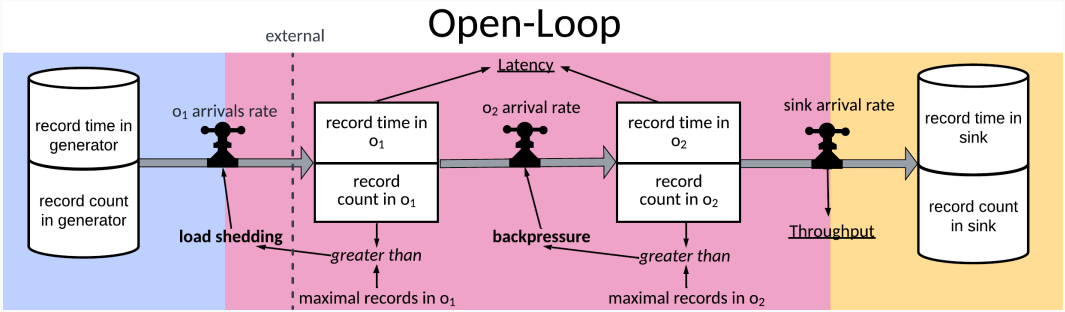



**Figure 6.** An open SPS benchmark operates in a similar way to the deployed case, with the generator continuing agnostic of how the SUT functions.

# Closed-loop SPS benchmarks are simpler to implement

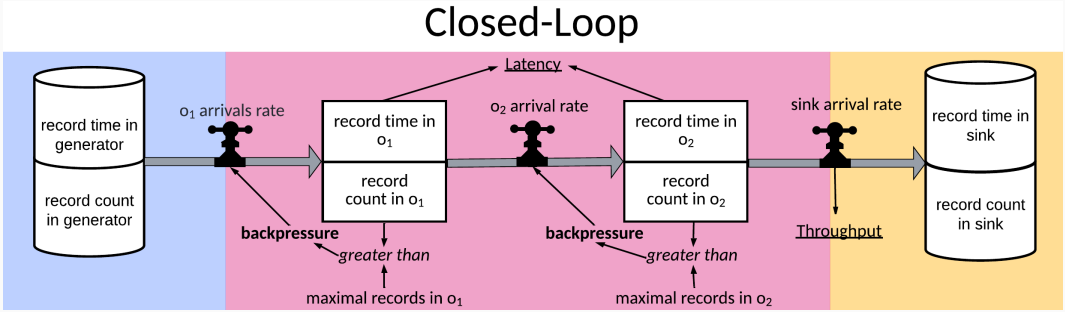



**Figure 7.** A closed SPS benchmark cannot ensure the level of separation between the generator and SUT, resulting in possible feedback if the SPS ingestion buffer overflows.

# Little evidence of backpressure in SPS benchmarking literature

Newcastle University


ALL (”backpressure” OR ”back pressure” OR ”back-pressure”)

AND ALL (”stream” OR ”streaming”)

AND ALL (”benchmark” OR ”benchmarking”)

AND NOT ALL (”vehicle”)

LIMIT-TO (SUBJAREA , ”COMP”)

LIMIT-TO (LANGUAGE , ”ENGLISH”)

Without Backpressure Limiter we received 52,620 articles, including it resulted in 63 articles as of 12th August, 2025.

| | Scopus | Alt BP | Citation Only |
|---|---|---|---|
| # Papers | 63 | 2 | 22 |
| | Public Access | | Final Included |
| # Papers | 1 | | 38 |

**Table 1.** Manual Review of Scopus Query. **Alt BP** = Excluded for being about an alternative definition of backpressure, **Citation Only** = Excluded for only having reference to backpressure in citations, **Public Access** = Excluded for not being public accessible.

# Further analysing results show

0. Papers that **do not explicitly acknowledge** backpressure in the text
1. Papers that explicitly acknowledge backpressure in the text but **do not monitor** for it during benchmarking
2. Papers that explicitly acknowledge backpressure in the text and monitors for it during benchmarking but **do not mitigate** the effects
3. Papers that explicitly acknowledge backpressure in the text, monitors for it during benchmarking, and mitigates the effects to prevent it from confounding results

| Level 0 | Level 1 | Level 2 | Level 3 |
|---|---|---|---|
| 22 | 32 | 7 | 2 (One of which is our previous publication) |

**Table 2.** Categorisation of literature into levels 0, 1, 2, and 3 of backpressure acknowledgement

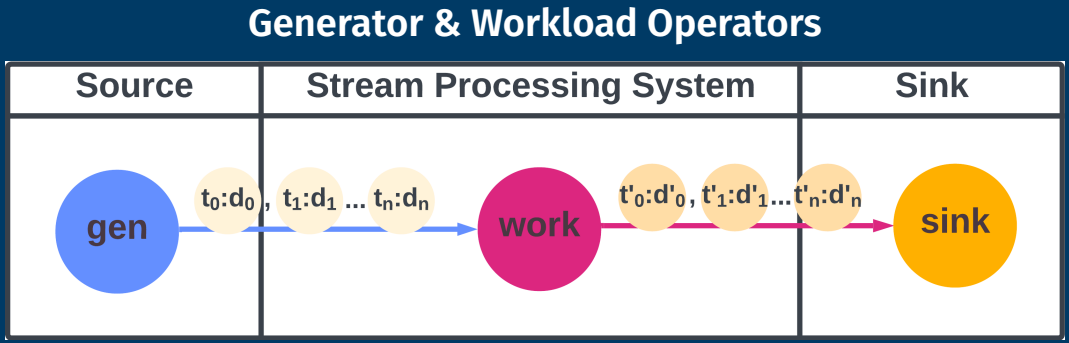



Figure 8. Experimental testbed framework.

# Generator operator creates $n$ records for $W$ windows

**Algorithm 1:** Basic Generation Loop

**Input:** $W, \Lambda, \delta_L = 1000$

1 **for** $w := 0$ *to* $W - 1$ **do**
2  $\lambda_w := \Lambda(w)$
3  $n_w := \lambda_w * \delta_L/1000$
4  $t_{start} :=$ TIME_MS()
5  **for** $i := 0$ *to* $n_w - 1$ **do**
6  rec:=GEN()
7  rec.gen_time:=TIME_MS()
8  SEND(rec)
9  $\delta_{transmit} :=$ TIME_MS() $- t_{start}$
10  $\delta_{remain} := \delta_L - \delta_{transmit}$
11  $\delta_w := \delta_{transmit} + \delta_{remain}$
12  **if** $\delta_{remain} > 0$ **then**
13  WAIT_MS($\delta_{remain}$)

| | |
|---|---|
| $W$ | Number of windows |
| $\delta_L$ | Target window duration |
| $w$ | Window index |
| $\Lambda$ | Arrival process |
| $\lambda_w$ | Target arrival rate |
| $n_w$ | Target arrival load |
| $t_{start}$ | Window start time |
| $\delta_{transmit}$ | Transmit duration |
| $\delta_{remain}$ | Reamining duration |
| $\delta_w$ | Window duration |
| rec.gen_time | Record Generation |

# Generator produced transmission windows by sending and waiting

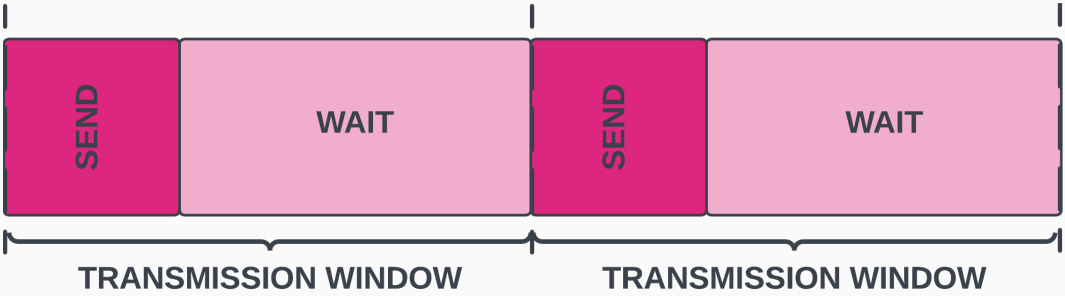



**Figure 9.** Algorithm 1 produces a transmission pattern where for each transmission window the generator sends all of the records, then waits until the next transmission window.

# Using BGL we can generate a variety of arrival processes

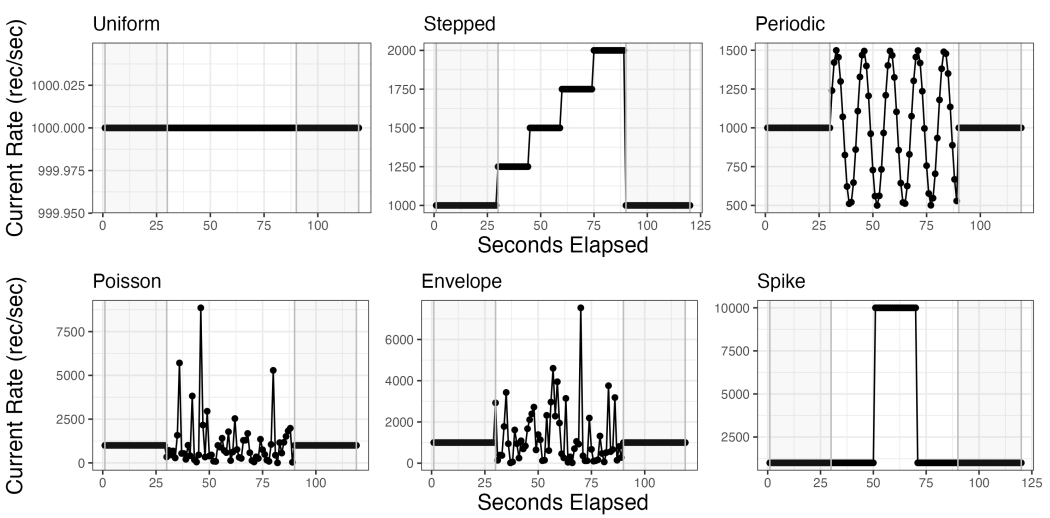



**Figure 10.** Implemented arrival processes using the experimental generator operator: uniform, periodic, poisson, envelope, and spike. For each process, the generator is provided with ($W = 60, \lambda = 1000$).

# Workload operator simulates $\delta_{work}$ every $f_{work}$ records

**Algorithm 2:** Workload Simulation Loop

**Input:** $\delta_{cost}, f_{work}, t_{start}, t_{end}$

1 $i := 0$
2 $t_{start}$ :=TIME_MS()
3 **while** $rec = GET\_NEXT()$ **do**
4 $\quad t_{current}$ :=TIME_MS()
5 $\quad j := (t_{current} - t_{start})/1000$
6 $\quad$ **if** $t_{current} \geqslant t_{act} \wedge t_{current} < t_{deact}$ **then**
7 $\quad\quad$ **if** $j \mod f_{wind} = 0 \wedge i \mod f_{work} = 0$ **then**
8 $\quad\quad\quad$ WAIT_MS($\delta_{cost}$)
9 $\quad\quad$ SEND(rec)
10 $\quad i := i + 1$

| | |
|---|---|
| $\delta_{cost}$ | Per-record cost |
| $f_{work}$ | Work Frequency |
| $t_{start}$ | Workload start time |
| $t_{end}$ | Workload end time |
| $t_{act}$ | Workload activation time |
| $t_{deact}$ | Workload deactivation time |
| $f_{wind}$ | Window Frequency |
| $i$ | Record Index |
| $j$ | Window Index |

# Using WSL we can simulate a variety of pipeline workloads

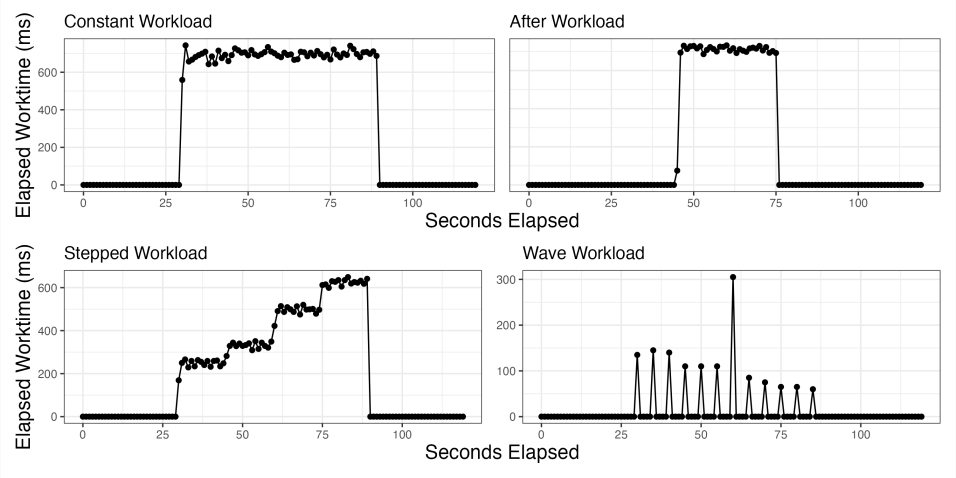



**Figure 11.** Implemented workload processes using the experimental workload operator: **constant workload** where $\delta_{cost}$ and $f_{work}$ are constant, **delayed workload** where the workload doe not activate immediately once the experiment starts, **stepped workload** where the workload increases in steps rather than remaining constant throughout the experiment, and **wave workload** where the work oscillates throughout the experiment.

# Triggering backpressure by escalating workload

Newcastle University

**Total workload ($\Delta$) experienced by the workload can be thought of as an product of the per-record cost ($\delta_{cost}$), work frequency ($f_{work}$), and per-second arrival rate ($\lambda_w$), such that:**

$$\Delta = \frac{\delta_{cost} \cdot \lambda_w}{f_{work}}$$

**By increasing total workload greater than 1000ms, we can fill the workload operator's input buffers and trigger backpressure in the generator, and by monitoring the transmission window duration detect it's occurrence.**

# Experiment parameters

We want to run a 60 second experiment, with 30 seconds of calibration (shown in highlighted grey) time pre and post experiment for the system to stabilise and investigate metrics either side of the experiment.

Any metric captured after the experiment and calibration windows have finished will be considered an **experimental overrun** (shown in highlighted dark red).

The generator operator will provide a constant arrival process with each transmission window targetting 1000ms, and workload operator producing a total workload totalling 500ms, 1000ms [1], and 2000ms.

[1] In practice this equates to slightly over 1000ms of workload due to logic overhead, producing a total workload which slightly exceeds the transmission window.

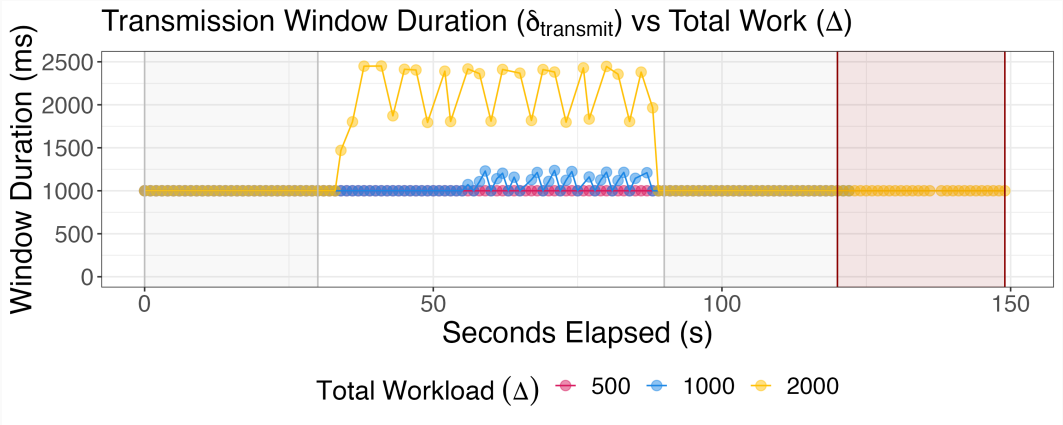



**Figure 12.** Transmission window durations ($\delta_{transmit}$) for increasing total workloads ($\Delta = 500, 1000, 2000$).

Newcastle University

### msBackpressuredPerSecond

”The time (in milliseconds) this task is back pressured (soft or hard) per second. It’s a sum of softBackPressuredTimeMsPerSecond and hardBackPressuredTimeMsPerSecond.” Rolling average over the past 60 seconds.

### isBackpressured

”Whether the task is back-pressured.” Per-second boolean measure if an operator is in the backpressured state.

### outPoolUsage

”An estimate of the output buffers usage. The pool usage can be > 100% if overdraft buffers are being used.” Per-second measure of the output buffer utilisation across all of an operators output buffers.

**Flink Metrics:** https://nightlies.apache.org/flink/flink-docs-master/docs/ops/metrics/

# Flink metrics provide Pro/Reactive Perspective of Backpressure

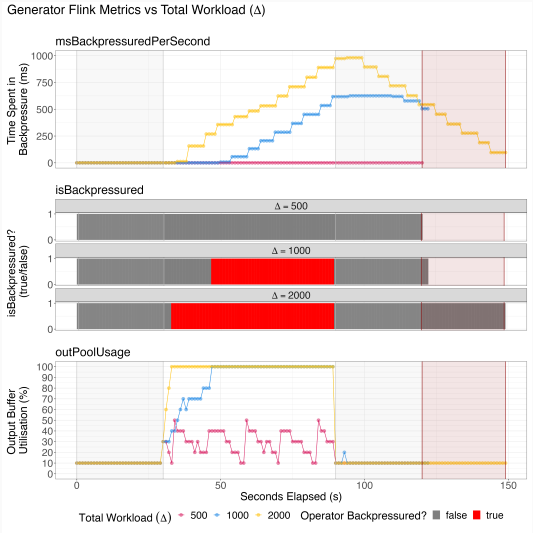
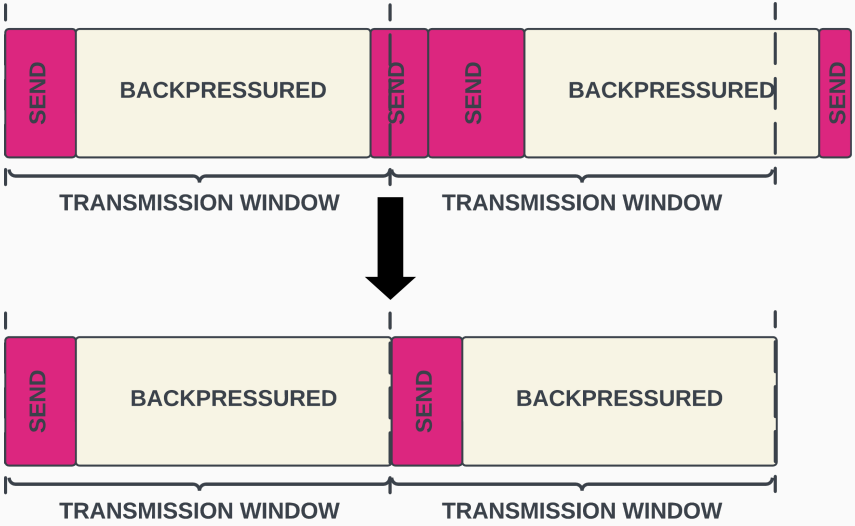



**Figure 13.** Flink metrics applied to escalating total workloads ($\Delta = 500, 1000, 2000$).

# Cutting off transmission windows when overrun



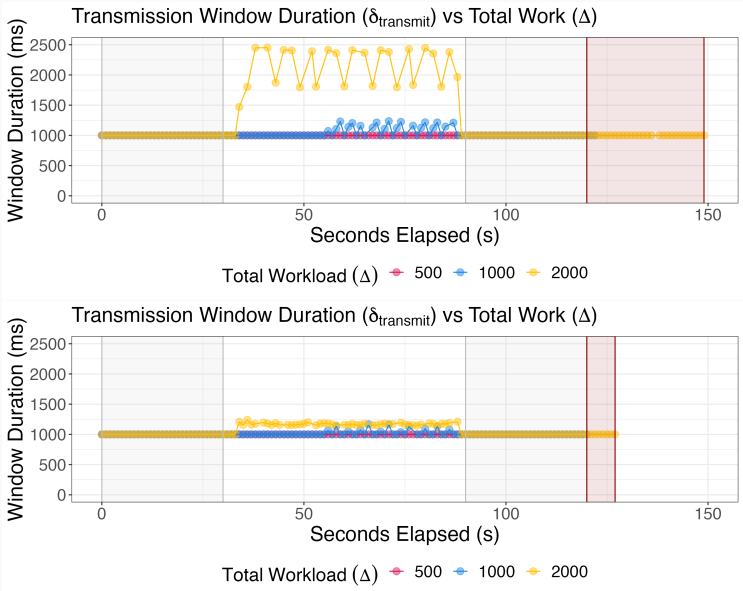
# Cutoff limits transmission window overruns

**Algorithm 3:** Cutoff

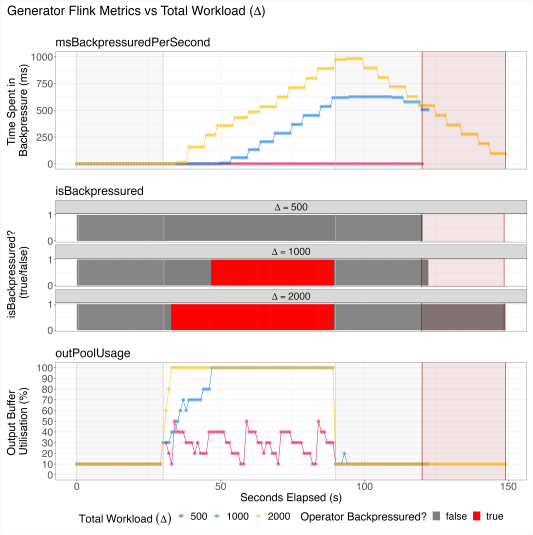
**Input:** $W, \delta_L, \lambda$

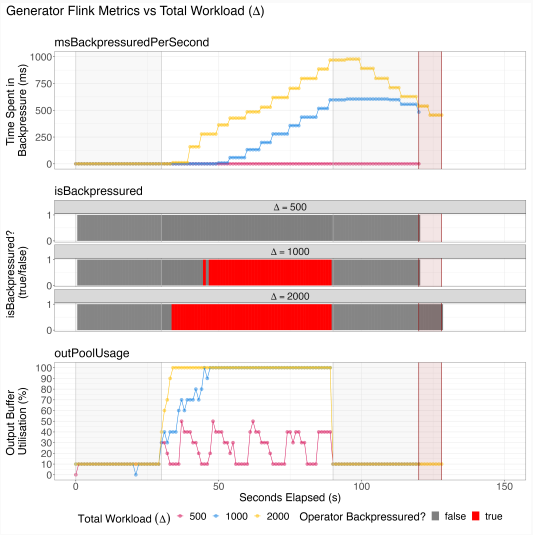
1 $\delta_{remain} := 0$
2 **for** $w := 0$ *to* $W - 1$ **do**
3 $\quad n_w := \lambda_w * \delta_L / 1000$
4 $\quad t_{start} :=$ TIME_MS()
5 $\quad$ **for** $i := 0$ *to* $n_w - 1$ **do**
6 $\quad\quad$ **if** *TIME_MS()*$-t_{start} > \delta_L$ **then**
7 $\quad\quad\quad$ BREAK()
8 $\quad\quad$ rec:=GEN()
9 $\quad\quad$ SEND(rec)
10 $\quad \delta_{transmit} :=$ TIME_MS() $-t_{start}$
11 $\quad \delta_{remain} := \delta_L - \delta_{transmit}$
12 $\quad \delta_w := \delta_{transmit} + \delta_{remain}$
13 $\quad$ **if** $\delta_{remain} > 0$ **then**
14 $\quad\quad$ WAIT_MS($\delta_{remain}$)
15 $\quad \delta_{remain} := \delta_{remain} + \delta_{wait}$

# Cutoff reactively reduces transmission duration overruns

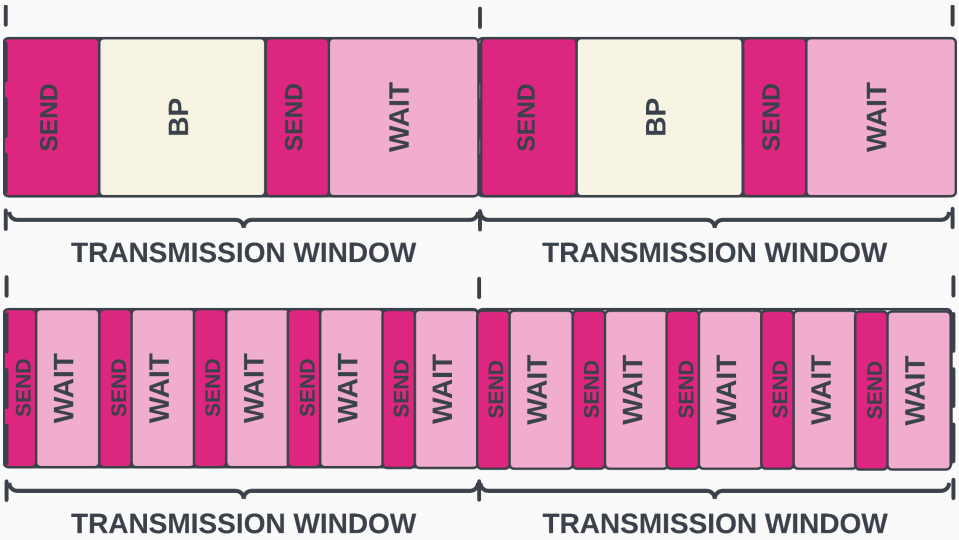


# Cutoff doesn't improve Flink metrics, but it does reduce overruns



# Spacing out records across a transmission mitigates backpressure



# Spaced records algorithm introduces spacing within transmission

Newcastle University

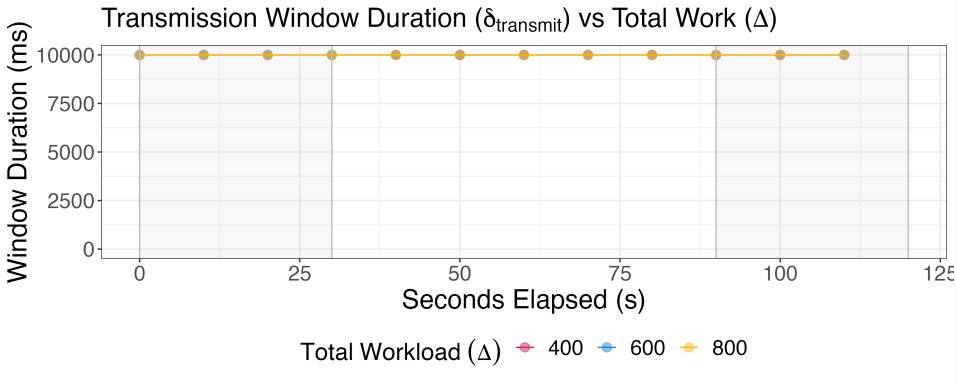
**Algorithm 4:** Spaced Out

**Input:** $W, \delta_L, \lambda, \delta_{inter}$

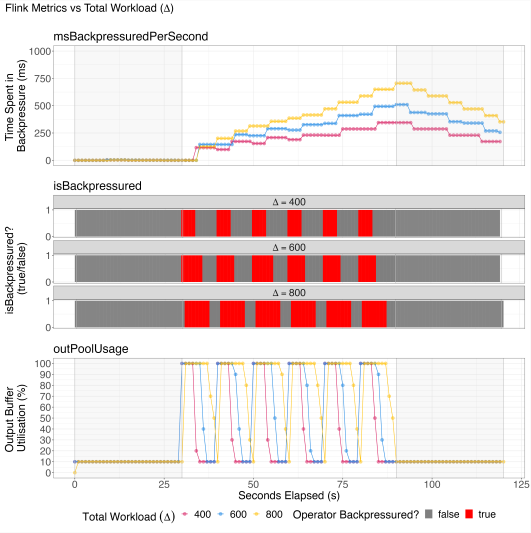
1 $\delta_{remain} := 0$
2 **for** $w := 0$ *to* $W - 1$ **do**
3 $\quad n_w := \lambda_w * \delta_L / 1000$
4 $\quad t_{start} :=$ TIME_MS()
5 $\quad$ **for** $i := 0$ *to* $n_w - 1$ **do**
6 $\quad\quad$ rec:=GEN()
7 $\quad\quad$ SEND(rec)
8 $\quad\quad$ **if** $\delta_{remain} > 0 \wedge i \mod \lambda_w / \delta_{remain} = 0$ **then**
9 $\quad\quad\quad$ WAIT_MS($\delta_{inter}$)
10 $\quad\quad\quad$ $\delta_{wait} := \delta_{wait} + \delta_{inter}$
11 $\quad \delta_{transmit} :=$ TIME_MS() $- t_{start}$
12 $\quad \delta_{remain} := \delta_L - \delta_{transmit}$
13 $\quad \delta_w := \delta_{transmit} + \delta_{remain}$
14 $\quad$ **if** $\delta_{remain} > 0$ **then**
15 $\quad\quad$ WAIT_MS($\delta_{remain}$)
16 $\quad \delta_{remain} := \delta_{remain} + \delta_{wait}$

| | |
|---|---|
| $\delta_{inter}$ | Interspersed wait duration |
| $\delta_{wait}$ | Total interspersed wait duration |

# Increase window to 10s to allow Flink metrics to capture backpressure

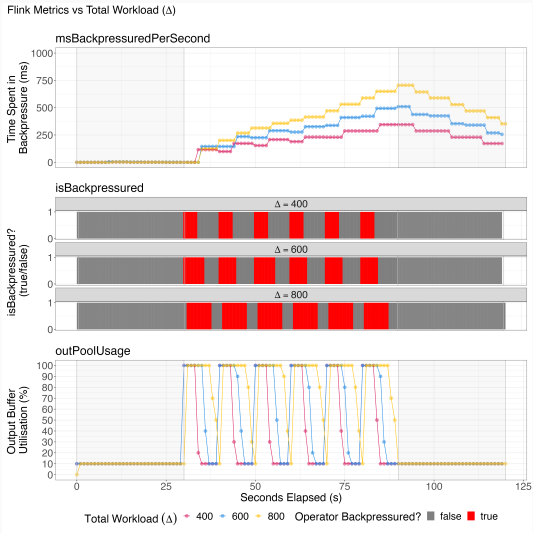


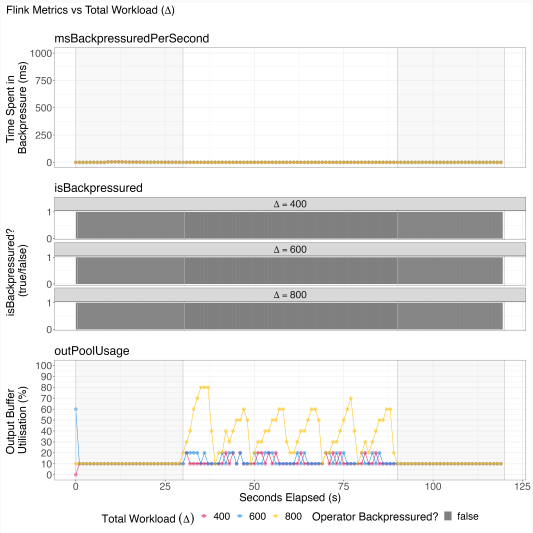
# Without spacing backpressure occurs within the 10s window



# Spacing mitigates backpressure which Flink cannot capture normally

Newcastle University



# Moderate benchmark results by thinking about benchmark model

Newcastle University

1. Is the benchmark closed-loop?
2. If so, is the generator affected by backpressure from the SUT?
3. If so, can you space records of cut off overrunning transmission windows?
4. If not, is the amount of backpressure captured in the metrics cause for concern? (Use-case specific)

# Thank You!

i.g.dixon1@ncl.ac.uk

# References

Kalavri, Vasiliki et al. **“Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows”.** In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. Cited by: 71. 2007, pp. 783–798.