iterflow: Composable Streaming Statistics for JavaScript

Gaurav Singh
Mathscapes Research
gv-sh@outlook.com

February 2026

Abstract

We present `iterflow`, a zero-dependency JavaScript library that integrates online statistical algorithms into lazy iterator pipelines. ES2025 Iterator Helpers provide lazy transforms but no statistics; libraries like `@stdlib/stats/incr` provide streaming accumulators but not as composable pipeline stages. `iterflow` bridges this gap: Welford's variance, EWMA, streaming Pearson correlation, z-score anomaly detection, and monotonic deque windowed extrema are implemented as chainable pipeline stages that compose with `map`, `filter`, `window`, and `take` while preserving lazy evaluation and $O(1)$ memory per stage. Benchmarks on synthetic workloads show a 296× throughput improvement over eager array pipelines with early termination at $N{=}10^6$, and a 780× improvement for streaming versus recomputed correlation at $N{=}10^4$. For standalone single-statistic computation, the generator-based design incurs 3–5× overhead relative to hand-written loops.

Keywords: streaming algorithms, iterator pipelines, online statistics, JavaScript, lazy evaluation

1 Introduction

A recurring pattern in server-side JavaScript is the need to compute running statistics over data too large or too incremental to materialise: monitoring dashboards processing millions of metrics, log analyzers scanning multi-gigabyte files, and sensor pipelines producing unbounded streams. Two capabilities are required. First, *lazy evaluation*: elements should be processed on demand, pipelines should terminate early, and intermediate arrays should not be allocated. Second, *streaming statistics*: mean, variance, correlation, and anomaly scores should be computable in a single pass with bounded memory.

JavaScript now offers partial solutions to each. ES2025 Iterator Helpers [9] added lazy `map`, `filter`, `take`, and `flatMap` to the iterator protocol, but no statistical operations; computing a windowed variance still requires collecting into an array. Conversely, `@stdlib/stats/incr` [8] provides high-quality standalone streaming accumulators—incremental mean, variance, covariance—but these are imperative objects, not composable pipeline stages. A workflow like "filter valid readings → sliding window → compute variance → threshold → take first 100" requires the user to manually iterate, manage circular buffers, feed each value into an accumulator, and implement early termination logic.

`iterflow` unifies lazy pipeline composition with streaming statistical algorithms. Its contribution is a *composition model* in which online algorithms are first-class pipeline stages: a pipeline `filter`→`window`→`streamingVariance`→`take` is expressed as a single chained expression, executes in constant memory, and terminates the moment the `take` count is satisfied.

The library is published as `@mathscapes/iterflow` on npm under the MIT license. It has zero runtime dependencies, full TypeScript type inference, and dual ESM/CJS builds. Source code and benchmarks are available at https://github.com/mathscapes/iterflow. DOI: 10.5281/zenodo.18610143.

Table 1: Capability matrix for JavaScript data processing libraries.

	`iterflow`	ES2025 Iter.	`@stdlib`	RxJS	simple-statistics
Lazy evaluation	•	•		•	
Streaming statistics	•		•		
Pipeline composition	•	•		•	
Synchronous batch	•	•	•		•
Windowed aggregation	•			•	
Early termination	•	•			
Zero dependencies	•	native			•

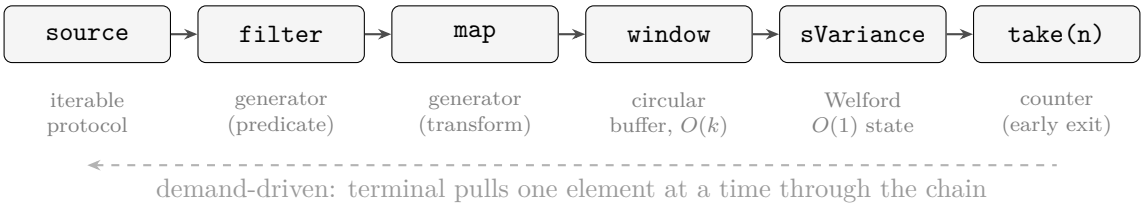

Figure 1: Architecture of a six-stage pipeline. Each stage is a generator function; the terminal (`take`) pulls elements through the chain one at a time. No intermediate arrays are allocated between stages. Stages maintain only their local state (e.g., circular buffer for `window`, three scalars for Welford's algorithm in `sVariance`).

2 Related Work

Table 1 summarises the landscape. No existing JavaScript library combines lazy pipeline composition with streaming statistics.

ES2025 Iterator Helpers [9] reached Stage 4 in 2025 and provide `.map()`, `.filter()`, `.take()`, `.drop()`, and `.flatMap()` on the iterator prototype. `iterflow` mirrors this API surface but extends it with windowing, chunking, and statistical operations. When transforms alone suffice, native helpers are preferable; `iterflow`'s value appears when statistical computation must compose within the pipeline.

`@stdlib/stats/incr` [8] provides streaming accumulators with careful numerical treatment. However, the accumulators are standalone objects: composing "filter → sliding window → variance → anomaly detection" requires imperative loops, manual buffer management, and bespoke early-termination logic. The composition burden falls entirely on the user.

RxJS [7] targets asynchronous reactive streams with concepts (Observables, Schedulers, backpressure) designed for event-driven I/O. It lacks built-in streaming statistical operators and introduces abstraction overhead unnecessary for synchronous batch analysis.

simple-statistics [6] provides a broad set of statistical functions operating on materialised arrays. It cannot process infinite or very large streams and does not support early termination.

3 Design

3.1 Pipeline composition via generators

The core abstraction is `Iterflow<T>`, a lightweight wrapper around JavaScript's iterable protocol. Each transform method—`map`, `filter`, `window`, `streamingVariance`, etc.—returns a new `Iterflow` backed by a generator function that lazily pulls values from the upstream source. No computation occurs until a terminal method (`toArray`, `reduce`, `first`, etc.) drives the iteration. Figure 1 illustrates the architecture.

Table 2: Time and space complexity of streaming operations. n = elements processed, k = window size.

Operation	Time	Space	Reference
`streamingMean`	$O(n)$	$O(1)$	eq. (1)
`streamingVariance`	$O(n)$	$O(1)$	Welford [10]
`ewma(`α`)`	$O(n)$	$O(1)$	Hunter [4]
`streamingCovariance`	$O(n)$	$O(1)$	Chan et al. [1]
`streamingCorrelation`	$O(n)$	$O(1)$	Chan et al. [1]
`streamingZScore`	$O(n)$	$O(1)$	eq. (6)
`windowedMin/Max`	$O(n)$	$O(k)$	Lemire [5]
`window(`k`)`	$O(n)$	$O(k)$	circular buffer
`variance` (terminal)	$O(n)$	$O(1)$	Welford [10]
`median` (terminal)	$O(n)$ avg.	$O(n)$	Hoare [3]

This design yields two properties. First, memory usage is $O(1)$ per stateless stage and $O(k)$ per windowed stage, independent of input size. Second, early termination propagates automatically: when `take(n)` stops pulling, upstream generators are never resumed, and unprocessed elements are never touched.

3.2 Type constraints

Statistical methods are constrained to `Iterflow<number>` via TypeScript method overloading. Calling `.sum()` on `Iterflow<string>` is a compile-time error, ensuring that streaming statistical stages receive numeric input without runtime type checks on the hot path.

4 Streaming Algorithms

Table 2 summarises the operations and their asymptotic complexity. All streaming transforms maintain bounded state and consume elements in a single pass.

Online mean and variance. Streaming variance uses Welford's algorithm [10], which avoids the catastrophic cancellation inherent in naïve two-pass or sum-of-squares formulations [2]. Given observations $x_1, \ldots, x_n$, the algorithm maintains a running mean M_k and aggregate squared deviation S_k:

$$M_k = M_{k-1} + \frac{x_k - M_{k-1}}{k} \tag{1}$$

$$S_k = S_{k-1} + (x_k - M_{k-1})(x_k - M_k) \tag{2}$$

Population variance at step k is $\sigma_k^2 = S_k/k$. The entire state consists of three scalars (k, M_k, S_k).

Exponentially weighted moving average. EWMA [4] computes a smoothed estimate that emphasises recent observations:

$$\mathrm{EWMA}_t = \alpha \cdot x_t + (1 - \alpha) \cdot \mathrm{EWMA}_{t-1}, \quad \alpha \in (0, 1] \tag{3}$$

with $\mathrm{EWMA}_0 = x_0$. The decay factor α controls responsiveness. State: one scalar.

Streaming covariance and Pearson correlation. The bivariate extension of Welford's method [1] computes covariance and correlation from paired streams $\{(x_i, y_i)\}$ in a single pass, maintaining means $\bar{x}_k, \bar{y}_k$, second moments $M_{2x,k}, M_{2y,k}$, and co-moment $C_{xy,k}$:

$$C_{xy,k} = C_{xy,k-1} + (x_k - \bar{x}_{k-1})(y_k - \bar{y}_k) \tag{4}$$

$$r_k = \frac{C_{xy,k}}{\sqrt{M_{2x,k} \cdot M_{2y,k}}} \tag{5}$$

Table 3: Filter-map-take(1000) pipeline throughput (ops/s). `iterflow` uses lazy generators; "Native array" uses `.filter().map().slice()`. Throughput RME below 1% for all measurements.

N	`iterflow`	Native array	Ratio
10^2	69,268	1,538,911	0.05×
10^3	9,812	143,443	0.07×
10^4	7,859	13,380	0.59×
10^5	7,917	434	18.2×
10^6	7,818	26	**296**×

State: six scalars. r_k yields the Pearson correlation coefficient at each step.

Streaming z-score. Anomaly detection via z-score uses the mean and standard deviation accumulated from *prior* observations:

$$z_k = \frac{x_k - M_{k-1}}{\sqrt{S_{k-1}/(k-1)}} \tag{6}$$

This pre-observation convention prevents the current value from influencing its own score. The first two elements yield NaN (insufficient data for a meaningful standard deviation).

Monotonic deque windowed extrema. Sliding-window minimum and maximum use monotonic deques [5] to maintain the current extremum in $O(1)$ amortised time per element. Each element enters and leaves the deque at most once, giving $O(n)$ total work with $O(k)$ space. This avoids the $O(nk)$ cost of naïve rescanning.

Circular buffer windowing. The `window(k)` transform writes each element at position $i \bmod k$ in a pre-allocated array, avoiding the $O(k)$ per-element cost of `shift()`-based naïve sliding windows.

Quickselect median. The terminal `median()` uses Hoare's quickselect [3] for $O(n)$ average-case selection, avoiding the $O(n \log n)$ cost of a full sort. NaN values are filtered before selection.

5 Evaluation

We evaluate `iterflow` on four benchmark suites using Tinybench on Node.js v22 (ARM64 Linux). All benchmarks are included in the repository and are reproducible via `node -import tsx benchmarks/*.bench.ts`. Data is synthetically generated; results report mean throughput over the default Tinybench iteration count.

5.1 Lazy evaluation with early termination

The primary motivation for `iterflow` is pipelines that terminate early on large inputs. Table 3 measures a filter→map→take(1000) pipeline as input size N varies.

The key observation is that `iterflow`'s throughput is effectively constant across N (∼8,000 ops/s), because only ∼1,000 elements traverse the full pipeline regardless of input size. Native array methods materialise the entire filtered and mapped result before slicing, degrading linearly. The crossover occurs between $N = 10^4$ and $N = 10^5$.

For small inputs ($N < 10^3$), the generator-based pipeline is slower due to per-element overhead from `yield`/`next()` dispatch. This is the expected trade-off: generators have higher constant factors than tight array loops, but their $O(1)$ work for early termination dominates at scale.

5.2 Streaming versus recomputed correlation

Table 4 compares `streamingCorrelation` (single-pass, $O(n)$) against a two-pass baseline that recomputes Pearson r from scratch at each step ($O(n^2)$ total).

Table 4: Streaming Pearson correlation over $N = 10{,}000$ paired observations.

Method	Lat. (ms)	Tput. (ops/s)
`iterflow` (1-pass)	0.66	1,514
Two-pass recompute	518	1.9
Ratio		**780**×

Table 5: Standalone statistical terminals, $N = 10^5$.

Terminal	`iterflow`	Naïve	Ratio
Mean	8,022	1,444	5.6×
Variance	1,815	691	2.6×

Table 6: Composition comparison: `iterflow` pipeline (6 LoC) vs. `@stdlib` imperative loop (12 LoC). Throughput in ops/s. Throughput RME below 1% for all measurements.

N	`iterflow`	`@stdlib` imper.	Ratio
10^3	9,699	91,477	0.11×
10^4	8,112	60,684	0.13×
10^5	9,705	87,870	0.11×

The improvement is algorithmic, not implementation-specific: $O(n)$ versus $O(n^2)$ total work. Any single-pass streaming implementation would show a similar advantage over the two-pass approach.

5.3 Statistical terminals

Table 5 compares `iterflow`'s terminal statistics against naïve JavaScript implementations on a full dataset with no pipeline composition overhead. Naïve mean uses `Array.reduce`; naïve variance uses two-pass sum-of-squares. Throughput RME below 1% for all measurements.

`iterflow`'s variance benefits from Welford's single-pass formulation, which avoids the second pass that the naïve implementation requires. The mean improvement arises from a plain `for-of` loop avoiding the per-element callback overhead of `Array.prototype.reduce`.

5.4 Generator overhead

For operations where `iterflow` wraps a single algorithm with no pipeline composition benefit, the generator machinery adds measurable overhead. EWMA at $N = 10^5$: `iterflow` achieves 213 ops/s versus 951 ops/s for a hand-written loop (4.5× overhead). Streaming z-score: 175 ops/s versus 594 ops/s (3.4×). This overhead is the per-element cost of the generator protocol (`yield`/`next()` plus closure allocation) and is inherent to the pipeline abstraction. It is fixed per element and amortised in multi-stage pipelines where the alternative—intermediate array materialisation or manual accumulator wiring—incurs far greater cost.

5.5 Composition comparison

To quantify the composition burden discussed in section 2, we compare `iterflow`'s pipeline against an imperative baseline using `@stdlib/stats-incr-mvariance` [8], a streaming moving-variance accumulator. The pipeline is: `filter`$(x > 50) \rightarrow$ `window(5)` $\rightarrow$ `variance` $\rightarrow$ `take(500)`.

The `@stdlib` imperative variant is ∼7–9× faster because `incrmvariance` maintains an $O(1)$ streaming accumulator per element, whereas `iterflow`'s `window(5).map(w => variance())` recomputes variance over each 5-element window. Both variants show near-constant throughput across N because `take(500)` terminates the pipeline early. The composition advantage is not performance but expressiveness: the `iterflow` pipeline is 6 lines of declarative code versus 12 lines of imperative filter/accumulate/terminate logic. When a streaming accumulator exists for the target statistic, the imperative approach will be faster; `iterflow`'s value is in pipelines that compose multiple stages without requiring bespoke loop construction for each combination. Note

that `incrmvariance` computes sample variance ($n-1$ denominator) while `iterflow`'s `variance()` computes population variance (n denominator); numerical results differ slightly but this does not affect the throughput comparison.

6 Discussion

Design decisions. `iterflow` has zero runtime dependencies; all algorithms are implemented directly, avoiding supply-chain risk and keeping the install footprint minimal. Each transform is a standalone generator function, not a class hierarchy; the `Iterflow<T>` class is a thin delegation layer. Statistical terminals compute population variance ($\sigma^2 = S_n/n$), not sample variance ($s^2 = S_n/(n-1)$), matching the typical use case of processing complete data streams. The streaming z-score uses statistics accumulated *before* the current observation, preventing the current value from influencing its own anomaly score—the correct convention for online detection.

Threats to validity. All benchmarks were run on a single platform (V8/Node.js on ARM64 Linux); generator overhead and JIT optimisation behaviour may differ on other engines (SpiderMonkey, JavaScriptCore) or architectures (x86-64). Workloads use synthetically generated random-walk data, which may not capture the branch-prediction and cache-locality patterns of real-world datasets. The `windowedMin`/`windowedMax` implementations use `Array.shift()` for front removal from the monotonic deque, which is $O(k)$ per operation rather than $O(1)$ amortised for a true deque; this is negligible for the small window sizes benchmarked ($k \leq 50$) but would degrade to $O(nk)$ total work for large k.

Limitations. The library targets synchronous iterables. Asynchronous stream support, streaming quantile estimation, and operator fusion for adjacent stateless stages are possible directions for future work.

7 Conclusion

`iterflow` provides a composition model in which streaming statistical algorithms are first-class stages in JavaScript's iterator protocol. It bridges ES2025 Iterator Helpers (lazy transforms without statistics) and standalone streaming accumulators (statistics without pipeline composition). Benchmarks confirm that the lazy pipeline model delivers order-of-magnitude improvements for multi-stage workflows with early termination or incremental computation, at the cost of constant-factor overhead for standalone single-statistic use. Source code, benchmarks, and documentation are available at https://github.com/mathscapes/iterflow.

References

[1] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. In *COMPSTAT 1982: Proceedings in Computational Statistics*, pages 30–41. Physica-Verlag, 1982.

[2] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

[3] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.

[4] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18(4):203–210, 1986.

[5] Daniel Lemire. Streaming maximum-minimum filter using no more than three comparisons per element. *Nordic Journal of Computing*, 13(4):328–339, 2006.

[6] Tom MacWright. simple-statistics: Statistical methods in readable JavaScript. https://github.com/simple-statistics/simple-statistics, 2024.

[7] RxJS Contributors. RxJS: Reactive extensions library for JavaScript. https://rxjs.dev, 2024.

[8] stdlib Authors. stdlib: A standard library for JavaScript and Node.js. https://github.com/stdlib-js/stdlib, 2024.

[9] TC39. ECMAScript iterator helpers proposal. https://github.com/tc39/proposal-iterator-helpers, 2025. Stage 4, ES2025.

[10] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.