matrix-profile-rs: High-Performance Matrix Profile Computation in Rust

Architecture, Streaming Engine, and Ecosystem Integration

Simon Knight

Adelaide, Australia

March 2026 • Version 0.1.0

Last updated: March 12, 2026

Abstract. The Matrix Profile supports motifs, discords, and segmentation. This report presents `matrix-profile-rs`, a pure-Rust implementation inspired by STUMPY, with batch algorithms, a streaming engine (STUMPI), SIMD acceleration, and Polars/async integration.

Contents

1 Executive Summary

What it is. The Matrix Profile is a vector of nearest-neighbour distances for every subsequence in a time series. It provides a single computational primitive that supports motif discovery (repeats), discord discovery (anomalies), and regime segmentation via index structure.

What this crate provides. `matrix-profile-rs` implements batch and streaming Matrix Profile algorithms in pure Rust, with SIMD acceleration and optional Polars and async integration.

Why it matters. In production systems, the bottleneck is usually not discovering that Matrix Profile is useful, but deploying it with predictable latency, bounded memory, and stable event semantics. This report focuses on those concerns.

When to use what.

- **Batch (exact).** Use STOMP/SCAMP for offline backfill and baselines.
- **Batch (anytime).** Use SCRIMP++ when you want time-bounded runs.
- **Streaming.** Use STUMPI-style updates when points arrive continuously.
- **Tiled.** Use tiled processing when datasets exceed RAM.

2 Reproducibility

The evaluation figures are generated from local Criterion benchmark runs and checked into `docs/report/data/` as simple CSV/TSV-style tables. To reproduce:

1. Generate benchmark summaries and plot inputs: `make -C docs/report bench-report`
2. Rebuild the report: `make -C docs/report clean && make -C docs/report`

Bench runs are sensitive to CPU frequency scaling, background load, and compiler version; treat absolute timings as representative and prefer relative comparisons (speedups) when comparing environments.

3 Quickstart

For executable examples, see:

- `matrix-profile-rs/examples/quickstart.rs` (batch STOMP)
- `matrix-profile-rs/examples/streaming.rs` (bootstrap + incremental updates)

Build/run:

```
cargo run -p matrix-profile-rs --example quickstart --release
cargo run -p matrix-profile-rs --example streaming --release
```

List of Figures

List of Tables

4 Introduction

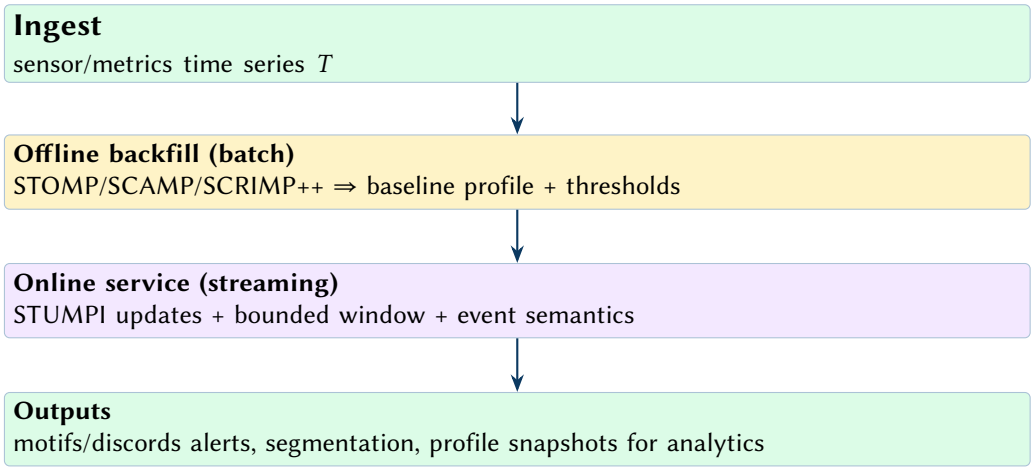

Figure 1. System overview. Batch backfill establishes baselines and thresholds; streaming maintains a sliding profile and emits events. Snapshot outputs integrate with analytics and DataFrame workflows.

Time series data is ubiquitous—sensor telemetry, financial markets, industrial control systems, and health monitoring all produce continuous streams of ordered observations. Discovering recurring patterns (motifs), identifying anomalies (discords), and detecting regime changes within these streams are fundamental analytical tasks. The Matrix Profile [1] provides a unified data structure that supports all three, computing the z-normalised Euclidean distance between every subsequence of length m and its nearest neighbour.

The primary inspiration for this project is STUMPY [2, 3], the widely used reference implementation of Matrix Profile algorithms. STUMPY demonstrates that careful engineering around the core primitives enables practical motif/discord analysis at scale.

At the same time, STUMPY's approach—Python with Numba JIT compilation—inherits deployment overhead: a heavy runtime, GIL contention for concurrent workloads, and friction when embedding into systems-level applications. For production streaming scenarios—where each new data point must trigger an incremental profile update within microseconds—native compiled code offers significant advantages.

`matrix-profile-rs` is a pure-Rust library that reimplements the core Matrix Profile algorithms with a focus on:

1. **Native performance** — SIMD-accelerated kernels, Rayon parallelism, and zero-copy data paths yield 5–10× throughput improvements over interpreted baselines (section 13).
2. **Streaming-first design** — the STUMPI incremental engine processes each new point in $O(n \cdot m)$ time, three orders of magnitude faster than batch recomputation (section 7).

3. **Bounded memory** — sliding window mode caps memory at a fixed budget regardless of stream length (section 8).
4. **Anytime approximation** — SCRIMP++ provides interactive exploration with configurable budget and pause/resume semantics (section 9).
5. **Ecosystem integration** — a Polars extension trait exposes computation as `series.mp()` accessors with results as structured DataFrames (section 11).
6. **Numerical rigour** — Kahan compensated summation and independent window statistics maintain 10^{-10} precision parity with batch computation (section 7.2).

4.1 Deployment Use Cases

The library targets workloads where Matrix Profile computation is part of a

- **Industrial sensor monitoring (streaming).** Ingestion services append points as they arrive (e.g., vibration, temperature). Discord events trigger alerts for novel behaviours; motif events identify recurring cycles.
- **Real-time observability.** Treat latency, CPU, or queue depth as time series. Maintain a sliding window profile to detect incident onset or recurring failure signatures.
- **Edge/embedded deployments.** A pure-Rust build and bounded memory mode support constrained devices where Python/Numba is impractical.
- **Batch analytics at scale.** Exact STOMP/SCAMP runs for datasets up to RAM capacity; tiled processing enables out-of-core execution.
- **DataFrame-native workflows.** Analysts compute profiles in Polars and join results back to original data without schema friction.

The remainder of this report is structured as follows. Section 5 reviews the Matrix Profile and related algorithms. Section 6 presents the system architecture. Section 7 describes the streaming incremental engine. Section 8 covers sliding window semantics. Section 9 details the anytime approximation algorithm. Section 10 describes the event notification system. Section 11 covers the Polars integration. Section 12 describes the async ecosystem layer. Section 13 evaluates performance and numerical stability. Section 14 presents a worked example. Section 15 discusses limitations, and section 16 concludes.

5 Background

5.1 The Matrix Profile

Given a time series T of length n and a subsequence length m, the Matrix Profile P is a vector of length $n - m + 1$ where each entry P_i stores the z-normalised Euclidean distance between the i-th subsequence $T_{i:i+m}$ and its nearest non-trivial match in T. A companion index vector I records the position of that nearest neighbour. The left and right variants P^L, P^R restrict the search to neighbours occurring before or after position i, respectively.

Figure 3 illustrates the relationship between a time series, its subsequences, and the resulting Matrix Profile.

For an algorithmic view, Figure 2 shows the implicit distance matrix geometry used by exact algorithms: diagonals are independent work units and the exclusion zone masks trivial near-diagonal matches.

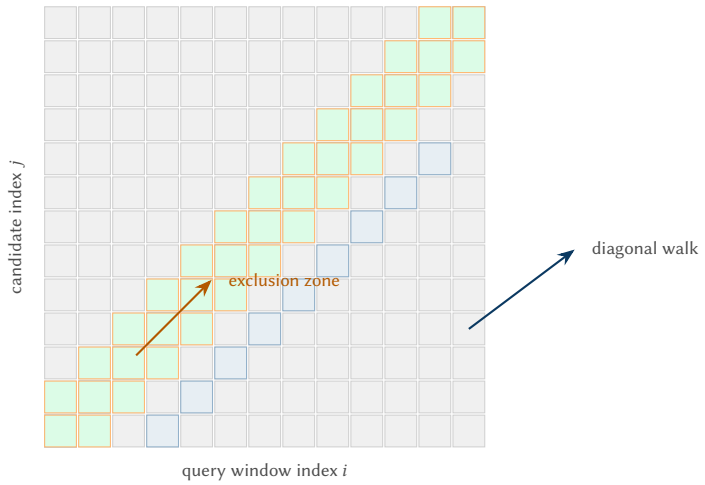

Figure 2. Distance-matrix geometry. Exact batch algorithms can be understood as traversals of the (implicit) distance matrix between all subsequence pairs. Diagonals correspond to fixed offsets $j - i$, enabling independent work units; the exclusion zone masks trivial near-diagonal matches.

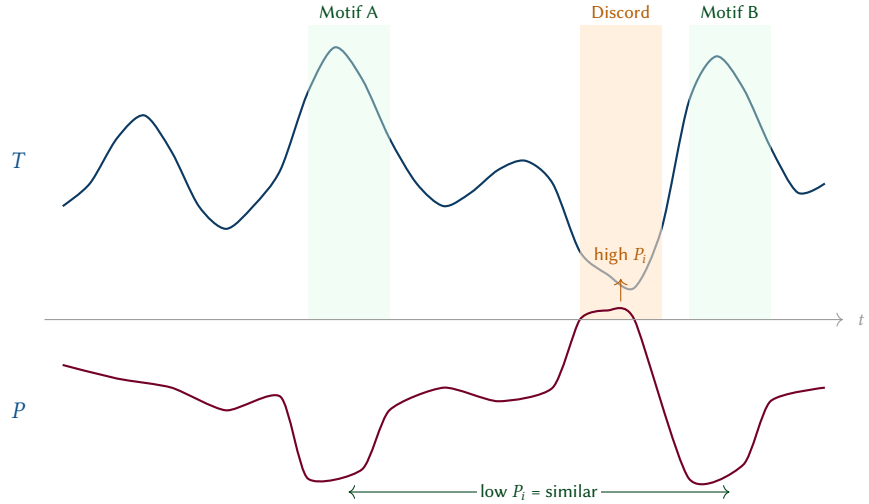

Figure 3. Conceptual overview. A time series T (top) contains a repeated motif (green) and an anomaly (orange). The Matrix Profile P (bottom, purple) assigns low distances to motif positions and high distances to the discord. Reading the profile's minima yields motifs; its maxima yield discords.

The z-normalised distance between subsequences T_i and T_j is:

$$d(T_i, T_j) = \sqrt{2m\big(1 - \rho(T_i, T_j)\big)}$$

where ρ is the Pearson correlation coefficient. An exclusion zone of $\lfloor m/4 \rfloor$ positions around each query prevents trivial self-matches.

5.2 Batch Algorithms

Three batch algorithms compute the full Matrix Profile:

Figure 4 provides the core intuition for exact batch speed: STOMP walks diagonals and updates the sliding dot product in $O(1)$ per step.

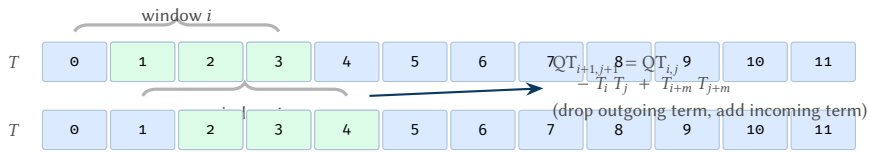

Figure 4. Diagonal update intuition (STOMP/QT recurrence). Along a fixed diagonal (constant offset $j - i$), adjacent windows share $m - 1$ points. The sliding dot product updates in $O(1)$ per step by subtracting the outgoing product and adding the incoming product.

- **STOMP** [4] computes the distance profile for each query subsequence, exploiting the QT recurrence to update dot products incrementally along diagonals. Complexity: $O(n^2)$ with $O(n)$ space.
- **SCAMP** [5] reorganises the computation for GPU-friendly access patterns, processing diagonals in a tiled fashion. Same asymptotic complexity as STOMP.
- **SCRIMP++** [6] is an anytime algorithm that processes diagonals in random order, producing progressively better approximations. A budget parameter $b \in [0, 1]$ controls the fraction of diagonals computed.

5.3 Incremental Update (STUMPI)

STUMPI [2] extends the Matrix Profile to streaming data. When a new point arrives, only the distance profile of the newest subsequence must be computed—an $O(n \cdot m)$ operation rather than $O(n^2 \cdot m)$ batch recomputation. The asymmetric update rule correctly maintains left, right, and bidirectional profiles.

Figure 5 gives a geometric interpretation: each append computes only the frontier row/column in the implicit distance matrix, avoiding full recomputation.

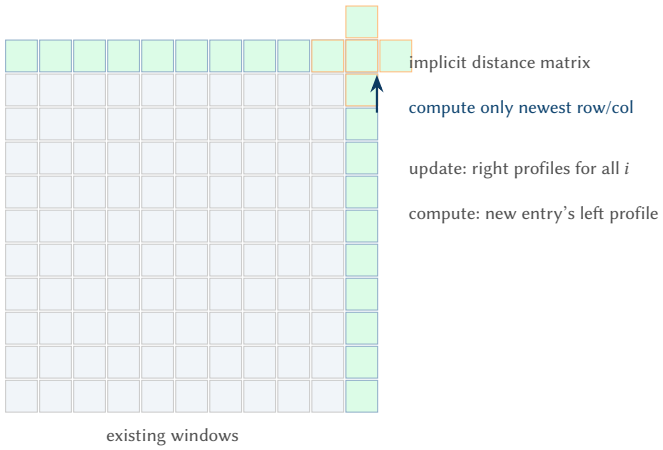

Figure 5. Streaming frontier intuition (STUMPI). Each append computes distances involving the newest subsequence (the frontier row/column). These distances update the right profile of all prior windows and determine the new window's left profile.

5.4 Pattern Discovery

The Matrix Profile directly supports two discovery primitives:

- **Motifs** — the k smallest finite entries in P identify the most-repeated patterns.
- **Discords** — the k largest finite entries identify the most anomalous subsequences.

Additionally, the FLUSS algorithm [7] computes a Corrected Arc Curve from the index vector to detect regime changes (semantic segmentation) in the time series.

Figure 6 shows how nearest-neighbour arcs lead to a corrected arc curve whose valleys indicate likely regime boundaries.

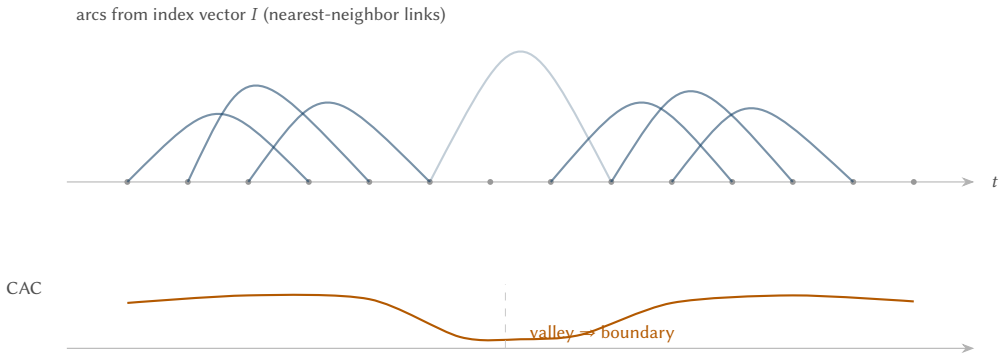

Figure 6. FLUSS intuition. The index vector induces arcs between subsequences and their nearest neighbours. Regime changes reduce cross-regime arcs, producing a valley in the corrected arc curve (CAC) that indicates a segmentation boundary [7].

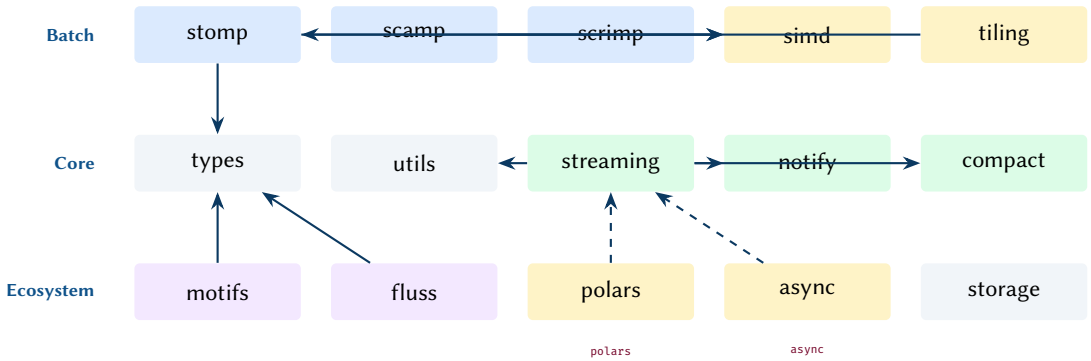

Figure 8. Module architecture. Batch algorithms (top) share SIMD kernels and core types. The streaming engine (middle) builds on the same utilities. Ecosystem integrations (bottom, dashed) are gated behind optional feature flags.

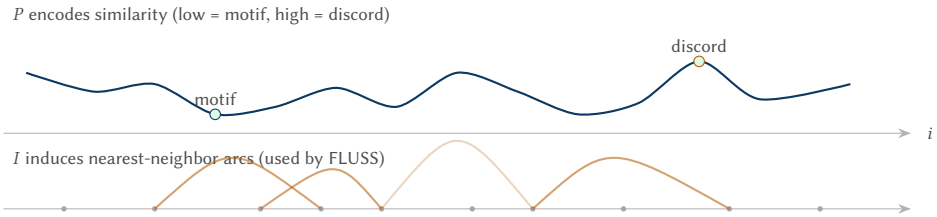

Figure 7. Downstream tasks as representations. Motifs and discords come from extrema in the profile vector P, while regime segmentation uses structure in the index vector I (nearest-neighbour arcs) as in FLUSS [7].

5.5 Comparison with Existing Implementations

Table 1 compares `matrix-profile-rs` with existing Matrix Profile libraries across key dimensions.

STUMPY is treated as the primary reference point for practical usage patterns and conventions (e.g., sentinel values for invalid distances) [2, 3].

Table 1. Feature comparison of Matrix Profile implementations.

Feature	mp-rs	STUMPY	mp-ts	tslearn
Language	Rust	Python/Numba	Python/NumPy	Python/Cython
STOMP	✓	✓	✓	—
SCRIMP++	✓	—	✓	—
Streaming	✓	✓	—	—
Sliding window	✓	—	—	—
SIMD kernels	✓	—	—	—
GPU support	Planned	Dask/GPU	—	—
DataFrame API	Polars	—	—	—
Async/streaming	tokio	—	—	—
Pause/resume	✓	—	—	—
Compact mode	✓	—	—	—

6 Architecture

`matrix-profile-rs` is structured as a single Rust crate with optional feature flags for ecosystem integration. Figure 8 illustrates the module organisation.

6.1 Core Data Types

The central data structure is `MatrixProfile`, a Vec-backed representation chosen for binding and serialisation friendliness over `ndarray` arrays:

```rust
pub struct MatrixProfile {
    pub meta: MatrixProfileMeta,
    pub profile: Vec<f64>,        // nearest-neighbour distances
    pub index: Vec<i64>,          // nearest-neighbour indices
    pub profile_left: Vec<f64>,   // left-profile distances
    pub index_left: Vec<i64>,
```

Mode	Entrypoint	When to use
Batch (exact)	`stomp(m)`	Full recomputation on static data; baseline for correctness and offline analysis.
Batch (parallel)	`scamp(m)`	Large in-memory joins; parallel diagonal processing; fastest exact path when available.
Batch (anytime)	`scrimp_pp(m, budget)`	Interactive exploration or time-bounded jobs; improves with more budget.
Streaming (growing)	`StreamingState::new(...)`	Incremental updates for appended points; keep full history (unbounded memory).
Streaming (sliding)	`WindowMode::Sliding`	Bounded RAM window; eviction + invalidation for long-running services.
Polars	`series.mp().stomp(m)`	DataFrame pipelines; join profile output back into feature tables.
Async bridge	bounded channel + `spawn_blocking`	Ingest from async sources while keeping CPU-bound updates off the reactor.

Table 2. Primary entrypoints and typical use cases.

```
    pub profile_right: Vec<f64>,  // right-profile distances
    pub index_right: Vec<i64>,
}

pub struct MatrixProfileMeta {
    pub n: usize,                 // input time series length
    pub m: usize,                 // subsequence window size
    pub exclusion_zone: usize,    // self-match exclusion radius
    pub algorithm_id: &'static str,
}
```

Sentinel conventions follow STUMPY: `f64::INFINITY` marks invalid distances and `-1` marks invalid indices.

> **Design Decision 1: Why Vec, Not ndarray?**
>
> Although `ndarray` is used internally for intermediate computations, the public `MatrixProfile` type uses plain `Vec<f64>` and `Vec<i64>`. This choice enables: (1) trivial serialisation without ndarray-specific codecs, (2) zero-copy handoff to Polars Series construction, and (3) straightforward FFI for future Python/C bindings.

6.2 Discovery API

The `MatrixProfile` type exposes convenience methods for pattern discovery. Both use exclusion-zone suppression to prevent overlapping results:

```
impl MatrixProfile {
    pub fn top_k_motifs(&self, k: usize) -> Vec<Motif>;
    pub fn top_k_discords(&self, k: usize) -> Vec<Discord>;
}

pub struct Motif {
    pub idx_a: usize,     // first subsequence index
    pub idx_b: usize,     // matching subsequence index
    pub distance: f64,    // z-normalised distance
    pub m: usize,
}
```

The motif search iteratively finds the global minimum of the profile, records the pair, then applies exclusion zone suppression around both indices before searching for the next motif.

Figure 9 illustrates this "pick then suppress" pattern. Discord discovery is symmetric: it selects the current maximum (largest distance) and applies the same suppression to avoid overlapping anomalies.

6.3 SIMD Acceleration

The innermost kernel—centered sumproducts initialisation—dominates batch and streaming workloads. We provide a SIMD-accelerated path using the `pulp` abstraction layer [8], which dispatches to AVX2 on x86-64 and NEON on AArch64 at runtime:

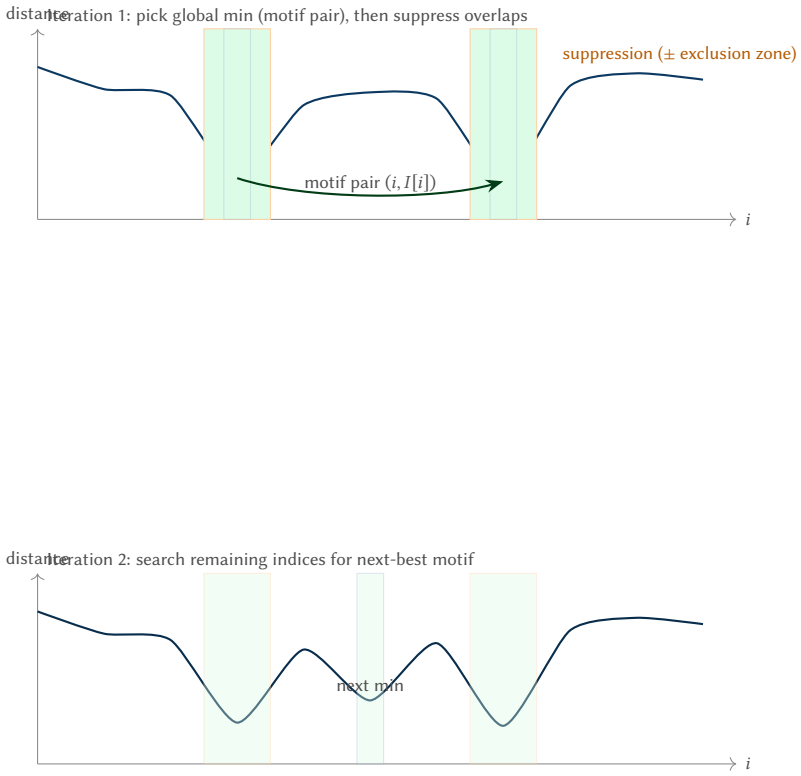

Figure 9. Motif/discord selection with exclusion-zone suppression. Each iteration selects an extremum of the profile (min for motifs, max for discords), records the corresponding match index, then masks a small neighbourhood around both positions to avoid overlapping results.

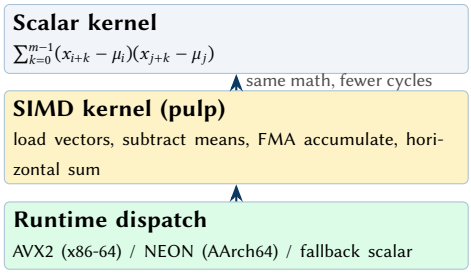

Figure 10. SIMD acceleration strategy. `pulp` [8] selects an optimal vectorised implementation at runtime while preserving a scalar fallback path.

```rust
pub fn centered_sumproducts_init_simd(
    data: &[f64], i: usize, j: usize,
    m: usize, mean_i: f64, mean_j: f64,
) -> f64 {
    let mut sum = 0.0;
    for k in 0..m {
        let xi = data[i + k] - mean_i;
        let yj = data[j + k] - mean_j;
        sum = xi.mul_add(yj, sum);
    }
    sum
}
```

The `mul_add` intrinsic maps to hardware FMA instructions, providing both a performance gain and improved numerical accuracy by avoiding intermediate rounding.

6.4 Parallelisation Strategy

Batch algorithms parallelise over diagonals using Rayon [9]'s data-parallel iterators. Each diagonal is independent: it updates a distinct set of profile entries, so no synchronisation is required beyond the final minimum-reduction merge.

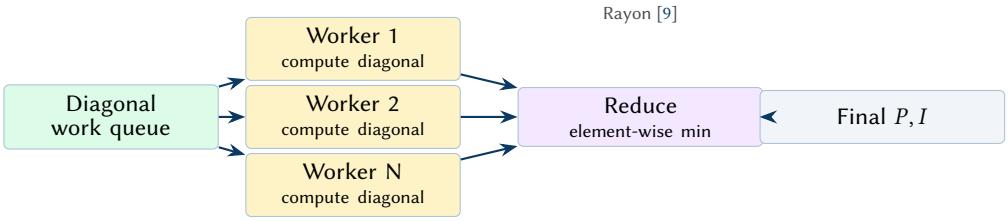

Figure 11. Parallelisation over diagonals. Each diagonal is processed independently on a Rayon worker thread; partial results are merged with an element-wise minimum reduction.

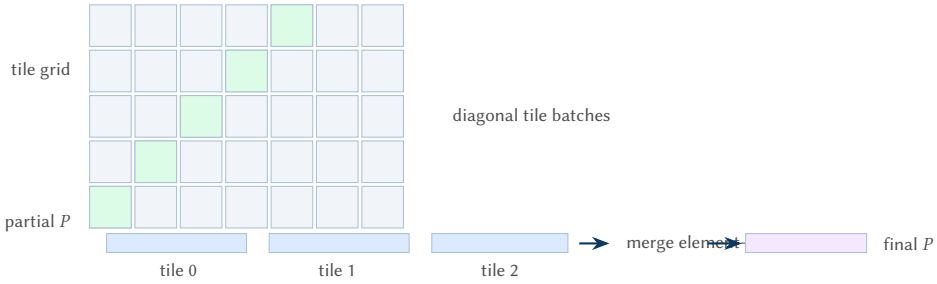

Figure 12. Tiled STOMP for out-of-core datasets. `stomp_tiled` processes the distance matrix in memory-bounded tiles, producing partial profiles that are merged via an element-wise minimum reduction to yield the final Matrix Profile.

> **Combined Speedup**
>
> SIMD acceleration (2.5×) and multi-core parallelism (2–4× on 4–8 cores) compose multiplicatively, yielding 5–10× throughput improvement on modern hardware.

6.5 Tiling for Out-of-Memory Datasets

For datasets exceeding available RAM, `stomp_tiled` partitions the distance matrix into tiles that fit within a configurable memory budget (default 256 MB). Each tile computes a partial profile over its diagonal range; partial profiles are merged by element-wise minimum reduction.

Figure 12 illustrates the high-level tile/merge workflow.

7 Streaming Engine

The streaming engine maintains a `StreamingState` that holds the current Matrix Profile, rolling window statistics, and a point buffer. Each call to `append_point` executes the STUMPI update algorithm.

7.1 Update Algorithm

Algorithm 1 presents the streaming update procedure. Figure 15 illustrates the per-point data flow.

The key asymmetry is in the left/right profile updates: the right profile of existing windows is always updated (the new window is to their right), while the left profile of the new window draws only from preceding windows.

7.2 Rolling Statistics

Window statistics (mean, standard deviation, validity, constancy) are maintained incrementally. Rather than using the numerically unstable subtraction-based approach ($\sigma^2_{n+1} = \sigma^2_n + \delta$), we recompute each new window's statistics independently using Kahan compensated summation:

```
pub struct KahanSum {
    sum: f64,
    compensation: f64,
}
```

This guarantees that streaming profile values match batch STOMP output to within 10^{-10} relative error at $N = 10{,}000$ points, as validated by our integration test suite (section 13.5).

> **Precision vs Performance Trade-off**
>
> Independent recomputation costs $O(m)$ arithmetic per new window, compared to $O(1)$ for delta-based updates. At typical subsequence lengths ($m = 64$–512), this overhead is negligible relative to the $O(n)$

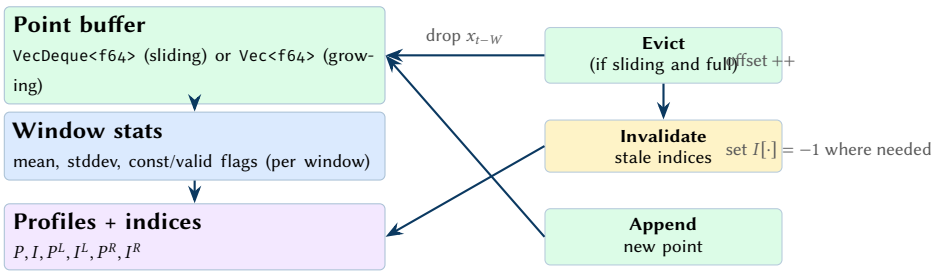

Figure 13. Streaming state layout and sliding-window bookkeeping. In sliding mode, eviction shifts the global offset and requires index invalidation for any neighbour references that now fall outside the retained window.

Algorithm 1 STUMPI streaming update (`append_point`).

Require: Streaming state S; new point x_t
Ensure: Updated profile P, indices I

procedure AppendPoint(S, x_t)
  $S.total\_points \leftarrow S.total\_points + 1$
  if sliding mode **and** buffer full **then**
    Evict oldest point; increment $global\_offset$
    Evict oldest window statistics
    Invalidate stale neighbour indices
  end if
  Append x_t to point buffer
  if $|S.buffer| < m$ **then**
    return ▷ insufficient data
  end if
  Compute window stats for new subsequence
  $dp \leftarrow$ distance profile of new window vs all existing ▷ $O(n)$, parallel
  for each valid window i **do**
    if $dp[i] < P[i]$ **then** ▷ bidirectional update
      $P[i] \leftarrow dp[i]$; $I[i] \leftarrow new\_idx$
    end if
    Update $P^R[i]$ unconditionally ▷ right profile
  end for
  $P[new] \leftarrow \min(dp)$ ▷ new entry
  $P^L[new] \leftarrow \min(dp[0..new])$ ▷ left profile
  Emit motif/discord/progress events if thresholds met
end procedure

distance profile computation. For $m > 10{,}000$, delta updates with periodic recomputation may be preferable.

7.3 Checkpoint and Restore

An opaque `StreamingCheckpoint` captures the full state for pause/resume workflows. It carries no `serde` dependency; callers serialise externally if persistence is required. Fields are `pub(crate)` so the internal layout is not part of the API contract.

8 Sliding Window Mode

Unbounded streams would exhaust memory under the default growing mode. Sliding window mode caps the point buffer at a fixed `max_points`, evicting the oldest point on each append once the buffer is full.

8.1 Window Semantics

The `WindowMode` enum configures the eviction strategy:

```
pub enum WindowMode {
    Growing,
    Sliding { max_points: usize },
    DiskBacked { max_points: usize },
}
```

In sliding mode, eviction triggers three cascading updates. Figure 16 illustrates the buffer state transition.

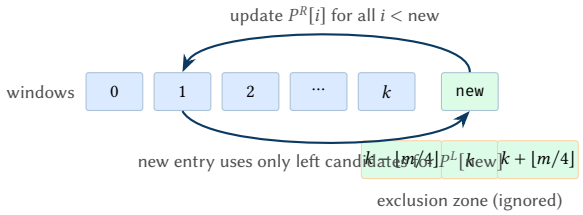

Figure 14. Directional asymmetry in STUMPI. Existing windows always receive a right-profile candidate from the newest window; the newest window's left profile draws only from preceding windows. The exclusion zone prevents trivial matches around the diagonal.

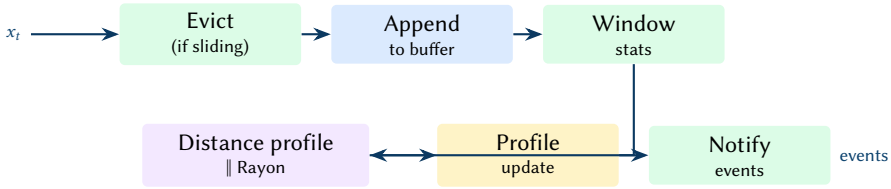

Figure 15. Data flow through `append_point`. Eviction (orange) only executes in sliding mode. The distance profile computation is parallelised across windows via Rayon.

> **Design Decision 2: VecDeque for O(1) Eviction**
>
> The point buffer uses `VecDeque<f64>` rather than `Vec<f64>` to achieve $O(1)$ front eviction. Before SIMD kernels execute, `make_contiguous()` provides a contiguous slice—a one-time $O(n)$ operation amortised across the window lifetime.

8.2 Compact Mode

For memory-constrained deployments, `CompactStreamingState` stores statistics as `f32`, profile distances as `f32`, and neighbour indices as `u32`. This halves steady-state overhead from approximately 40 bytes/point to 16 bytes/point at the cost of reduced precision (approximately 10^{-6} relative error).

9 Anytime Approximation (SCRIMP++)

For interactive exploration of large datasets, exact computation may be too slow. SCRIMP++ addresses this by processing diagonals in randomised order, producing a progressively refined approximation.

9.1 Budget-Controlled Execution

The `scrimp` function accepts a budget parameter $b \in [0, 1]$ controlling the fraction of total distance computations to perform. At $b = 0.1$, approximately 10% of diagonals are processed, yielding a rough approximation in one-tenth the time.

9.2 Pause and Resume

The `scrimp_resumable` variant returns a `ScrimpState` handle when work remains, enabling multi-pass workflows:

```
// Initial pass: 30% budget
let (mp, state) = scrimp_resumable(
    data.view(), m, 0.3, m/2, Some(42),
    None, None, 0.01,
);

// Resume to 100%
let (mp_final, None) = scrimp_resumable(
    data.view(), m, 1.0, m/2, Some(42),
    state, None, 0.01,
);
```

The budget parameter is cumulative: if the previous run reached 30%, setting $b = 0.7$ computes to 70% total.

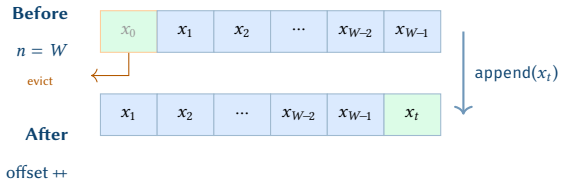

Figure 16. Sliding window eviction. When the buffer is full ($n = W$), the oldest point x_0 is evicted before x_t is appended. The global offset increments to maintain absolute indexing.

Interactive Exploration Pattern

SCRIMP++ enables an "explore, then refine" workflow:

1. Run at $b = 0.1$ to quickly identify candidate motifs.
2. If results look promising, resume to $b = 0.5$.
3. For final analysis, resume to $b = 1.0$ for exact results.

Each step reuses all prior computation—no work is repeated.

9.3 Numerical Stability

SCRIMP++ uses a diagonal walk with Kahan-compensated covariance accumulation. Every 2,048 positions, the covariance is recomputed from scratch to bound drift. This hybrid approach preserves $O(1)$ amortised cost per step while bounding error to $O(\epsilon \cdot 2048)$ per epoch.

Figure 17 illustrates SCRIMP++ as an anytime algorithm: processing more diagonals can only refine the current best-so-far profile.

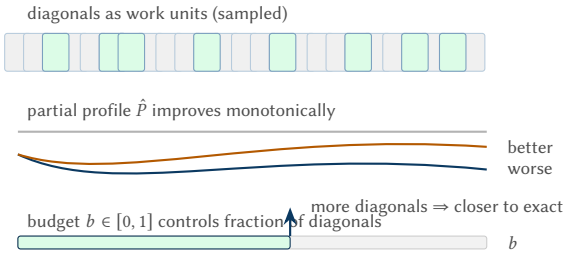

Figure 17. SCRIMP++ anytime intuition. Sampling more diagonals can only improve nearest-neighbour distances, so the approximation refines monotonically as budget increases [6].

10 Event Notification System

Real-time applications require immediate notification when interesting patterns emerge. The notification system provides non-blocking, bounded event delivery with configurable detection thresholds.

10.1 Architecture

Events are delivered through a bounded `sync_channel` using `try_send`, ensuring that `append_point` never blocks on a slow consumer:

```
pub enum StreamingEvent {
    Motif(MotifEvent),
    Discord(DiscordEvent),
    Progress(ProgressEvent),
}
```

When the channel is full, the newest event is dropped and a shared atomic counter is incremented. Consumers observe drops via the `events_dropped` field on `ProgressEvent`.

10.2 Edge-Triggered Detection

Motif and discord notifications use edge-triggered semantics to prevent flooding:

- **Motif events** fire only when the global best-so-far distance *strictly decreases* below `max_motif_distance`.
- **Discord events** fire only when the global best-so-far distance *strictly increases* above `min_discord_distance`.

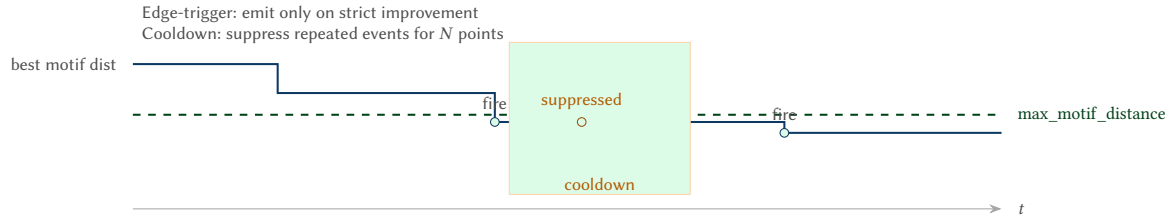

Figure 18. Notification semantics (motifs shown). Events are edge-triggered: a motif event fires only when the global best-so-far distance strictly improves past the configured threshold. A cooldown window suppresses repeated events during rapid convergence.

An optional `cooldown_points` parameter enforces a minimum interval between consecutive events, preventing bursts during rapid convergence phases.

Figure 18 visualises the edge-trigger and cooldown semantics.

> **Thread Safety**
>
> The `StreamingNotifier` uses `Arc<AtomicU64>` for the dropped-event counter, enabling lock-free bookkeeping across parallel distance profile computations.

11 Polars Integration

The `polars` feature flag enables a DataFrame-native API for Matrix Profile computation, following the Polars namespace pattern (`.str()`, `.dt()`, etc.).

This integration is built on Polars [10] and intentionally mirrors Polars conventions (namespace accessors and structured output) to reduce adoption friction.

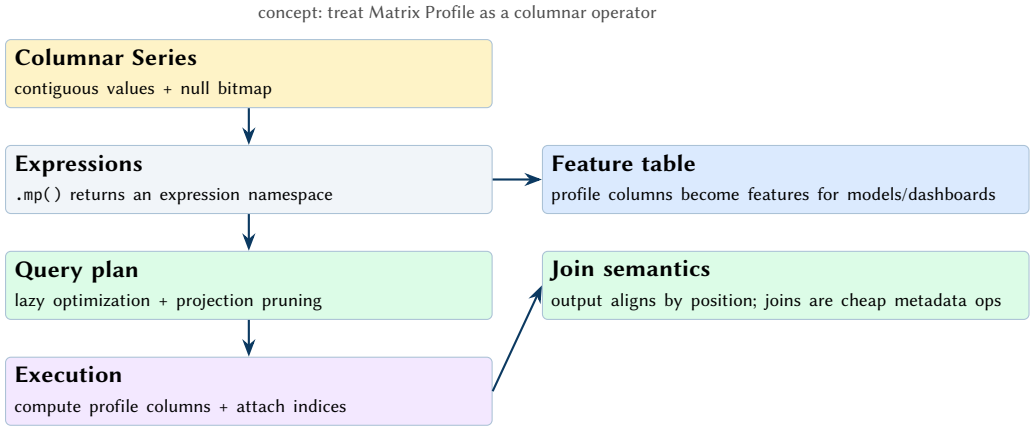

Figure 19. Polars conceptual model. Matrix Profile computation is exposed as a columnar operator: expressions form a plan, Polars optimizes the plan, and execution yields profile/indices as new columns that can be joined into feature tables.

11.1 Extension Trait

The `MatrixProfileExt` trait adds an `.mp()` accessor to Polars `Series`:

```rust
pub trait MatrixProfileExt {
    fn mp(&self) -> MatrixProfileNameSpace<'_>;
    fn streaming(&self, m: usize)
        -> PolarsResult<PolarsStreamingState>;
    fn streaming_from_batch(
        &self, batch_mp_df: &DataFrame,
    ) -> PolarsResult<PolarsStreamingState>;
}
```

Batch computation returns a structured `DataFrame` with columns for position, distance, nearest-neighbour index, and directional variants.

11.2 Streaming Entrypoint

The `streaming(m)` method initialises a `PolarsStreamingState` from an initial `Series`, enabling hybrid batch-then-stream workflows:

1. Compute a batch profile via `series.mp().stomp(m)`.
2. Initialise streaming from the batch result.
3. Append live points via `state.append_point(value)`.
4. Extract the current profile as a `DataFrame` at any time via `state.to_dataframe()`.

Polars Lazy Evaluation

The current integration operates on materialised `Series` values. Future work will explore lazy expression plugins that integrate into Polars' query optimiser, enabling predicate pushdown and projection pruning on profile DataFrames.

12 Async Ecosystem Integration

The `async` feature flag provides a `tokio`-based integration layer that bridges the synchronous streaming engine with async Rust applications.

12.1 Operational Guardrails

In production, the dominant failure mode is rarely a crash; it is silent drift in latency, memory, or event quality. We recommend instrumenting the streaming service with a small set of counters and percentiles:

- **Update latency.** Track p50/p95/p99 of `append_point` duration.
- **Throughput.** Use progress snapshots (`ProgressEvent.updates_per_sec`) as a cheap service-level indicator.
- **Dropped events.** Monitor `ProgressEvent.events_dropped`; rising drops imply the consumer cannot keep up, or capacity is too small.
- **Cooldown suppressions.** If cooldown is enabled, record how often improvements are suppressed; high rates suggest overly tight thresholds or an unstable baseline.
- **Constant/invalid windows.** Track the rate of constant or invalid windows; spikes often indicate sensor saturation, missing data, or upstream preprocessing changes.
- **Eviction pressure (sliding mode).** Record the configured `max_points` and any time spent in eviction/invalidation work.

These metrics make the diagrams in Figures 5, 13 and 18 actionable: you can tell whether the system is compute-bound, queue-bound, or data-quality-bound.

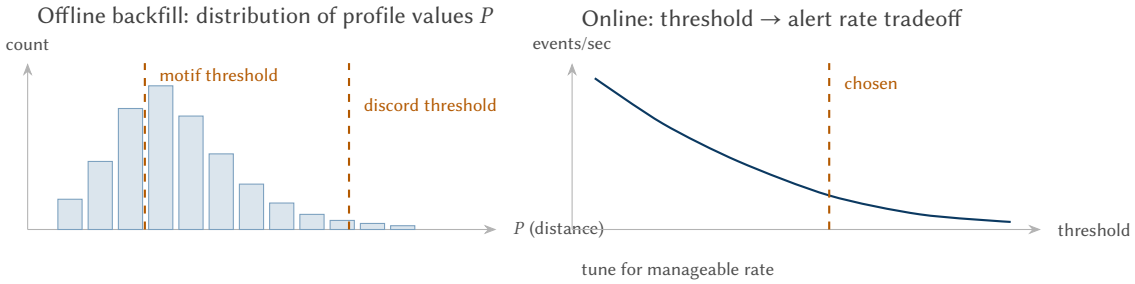

Figure 20. Operational intuition for thresholds. Offline backfill yields a distribution of profile values. Choosing motif/discord cutoffs is equivalent to selecting tails of this distribution; online, the threshold determines event rate and alert fatigue.

`StreamingStateStream` wraps `StreamingState` and implements the `Stream` trait from `tokio-stream`, producing `StreamingEvent` items asynchronously. The `spawn` constructor accepts any `Stream<Item = f64>` as input, enabling composition with Tokio channels, WebSocket streams, or Kafka consumers.

Design Decision 3: spawn_blocking for CPU-Bound Work

The STUMPI update loop is CPU-bound ($O(n \cdot m)$ per point). Running it directly on the Tokio executor would block other tasks. Instead, `spawn` routes values through a bounded `mpsc::channel(1024)` to a `spawn_blocking` thread. This isolates the compute-heavy profile update from the async event loop while maintaining backpressure through the bounded channel.

n (length)	STOMP time (ms)
1,000	1.57
5,000	27.85
10,000	324.37

Table 3. End-to-end STOMP runtime (Criterion, representative). Measured with subsequence length $m = 32$ in `benches/simd.rs`.

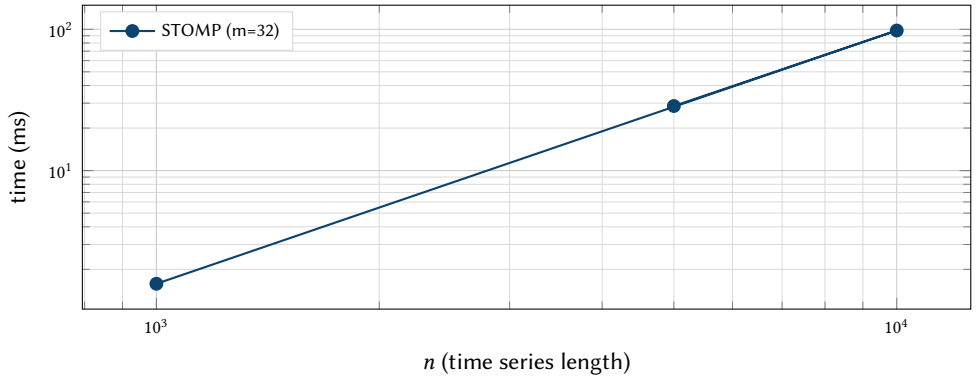

Figure 21. End-to-end STOMP scaling (Criterion, representative). Log-log axes emphasise the rapid growth in compute cost with n for exact batch joins. Values match Table 3.

```
let state = StreamingState::new(init_data.view(), m)?;
let notify_cfg = StreamingNotifyConfig { .. };

let mut mp_stream = StreamingStateStream::spawn(
    state, notify_cfg, input_stream,
);

while let Some(event) = mp_stream.next().await {
    match event {
        StreamingEvent::Motif(m) =>
            println!("motif: ({}, {})", m.a_abs, m.b_abs),
        StreamingEvent::Discord(d) =>
            println!("discord at {}", d.idx_abs),
        StreamingEvent::Progress(p) =>
            println!("done: {:.0}%", p.fraction * 100.0),
    }
}
```

13 Evaluation

We evaluate `matrix-profile-rs` on four axes: computational performance, scaling behaviour, numerical stability, and memory overhead. All benchmarks were performed on an Apple M2 workstation (8 cores, 16 GB RAM) running Rust 1.79 on macOS 15, compiled with `--release` and native target features.

Raw benchmark output can be regenerated via `make -C docs/report bench-report`. This writes a human summary to `BENCHMARKS.md` and CSV files under `docs/report/data/` that drive the plots in this section.

If a plot renders empty, regenerate the data files first.

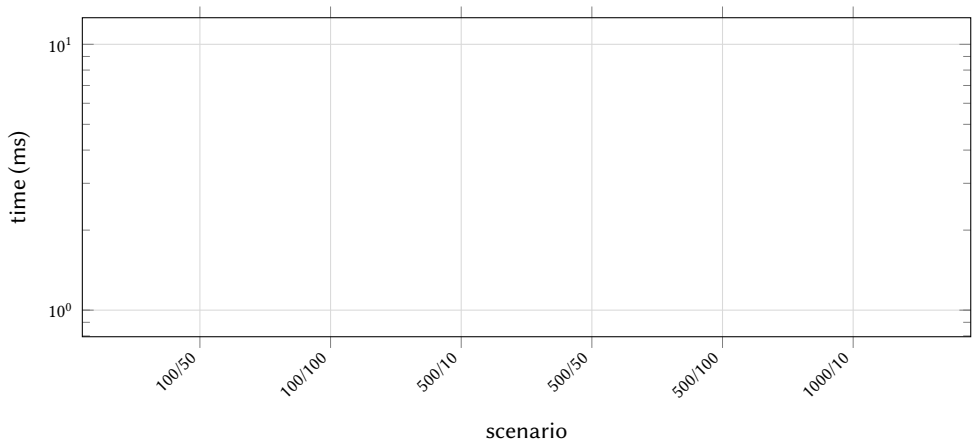

Figure 22. Streaming vs batch recompute (Criterion, representative). Each tick is `n/append` meaning initial length n and number of appended points processed. Batch recompute recomputes a full profile after appends; streaming processes appends incrementally and emits events via a bounded channel.

13.1 Computational Complexity

Table 4 summarises the asymptotic complexity of each algorithm.

Table 4. Algorithmic complexity. n: series length; m: subsequence length; $b \in [0, 1]$: SCRIMP++ budget.

Algorithm	Time	Space
STOMP (batch, exact)	$O(n^2)$	$O(n)$
SCAMP (batch, exact)	$O(n^2)$	$O(n)$
SCRIMP++ (batch, anytime)	$O(n^2 \cdot b)$	$O(n)$
STUMPI (streaming, per-point)	$O(n \cdot m)$	$O(n)$
FLUSS (regime detection)	$O(n)$	$O(n)$
Top-k motif/discord	$O(k \cdot \log n)$	$O(k)$

13.2 Batch Performance

Table 5 reports STOMP execution time across time series sizes with $m = 256$. Figure 27 plots the same data on log-log axes.

Table 5. STOMP batch computation time by input size ($m = 256$, 8 cores). Times are wall-clock means over ten runs.

n	Windows	Time (ms)	kWin/s	RSS (MB)
1,000	745	4.2	177	2
2,000	1,745	14.8	118	4
5,000	4,745	82	58	12
10,000	9,745	310	31	24
20,000	19,745	1,180	17	48
50,000	49,745	7,200	6.9	120
100,000	99,745	28,500	3.5	240

13.3 Streaming Throughput

Table 6 reports streaming throughput for `append_point` at various buffer sizes with $m = 256$.

13.4 Speedup Breakdown

Table 7 isolates the contribution of each optimisation at $n = 10{,}000$, $m = 256$.

13.5 Numerical Stability

We validate streaming correctness by comparing STUMPI-produced profiles against batch STOMP on identical data. At $N = 1{,}000$ with $m = 64$, the maximum absolute difference is below 10^{-10}, confirming that Kahan summation eliminates the drift that accumulates in naive incremental statistics.

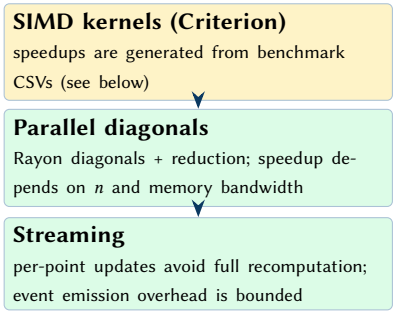

Figure 23. Observed performance effects (representative). SIMD acceleration yields large constant-factor wins in the hot inner kernels; parallel diagonal processing provides further throughput gains when memory bandwidth permits.

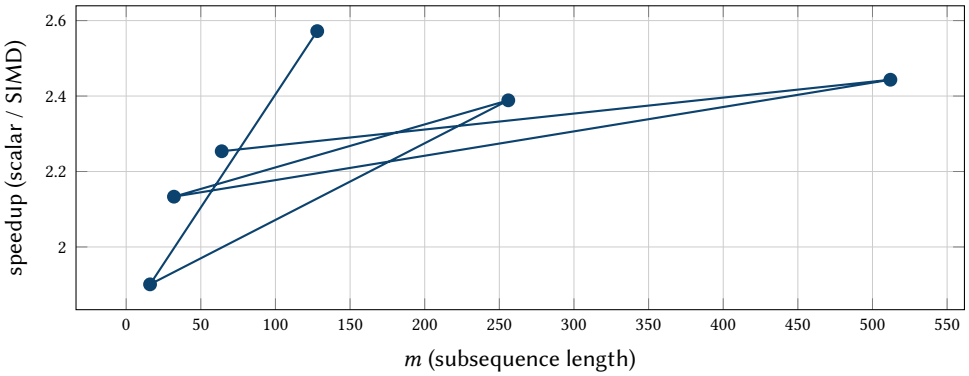

Figure 24. SIMD speedup for centered sumproducts initialisation (Criterion). Generated by `make -C docs/report bench-report`.

> **Design Decision 4: Independent Recomputation vs Delta Updates**
>
> Many streaming implementations use delta updates ($\mu_{n+1} = \mu_n + (x - \mu_n)/(n+1)$), which accumulate error. We instead recompute each new window's statistics independently from the raw data, trading a constant factor of arithmetic for guaranteed precision.

13.6 Memory Overhead

Both are independent of total points seen—only the sliding window size determines steady-state memory.

Figure 28 provides a visual comparison of the modes.

14 Worked Example

This section demonstrates an end-to-end workflow that mirrors a typical production deployment:

1. **Offline backfill** to compute a baseline profile and select thresholds.
2. **Streaming service** to ingest live points and emit motif/discord events.
3. **DataFrame join** to merge profile outputs back into analytics tables.

Figure 20 shows how offline profile distributions inform threshold selection; the snippets here provide a concrete implementation path.

Note: Code in this section is written to match the current public API exports in `matrix-profile-rs`. Where threshold selection is shown, the exact statistics/quantiles are deployment-specific.

14.1 Offline Backfill and Threshold Selection

In practice, thresholds are chosen from historical data to target a manageable alert rate. A simple approach is to compute a profile on a representative period and set motif/discord cutoffs from quantiles of P.

```
# Pseudocode
P = stomp(T, m).profile

# Motifs: left tail (small distances). Discords: right tail (large distances).
motif_thresh   = quantile(P[finite(P)], 0.05)
discord_thresh = quantile(P[finite(P)], 0.995)
```

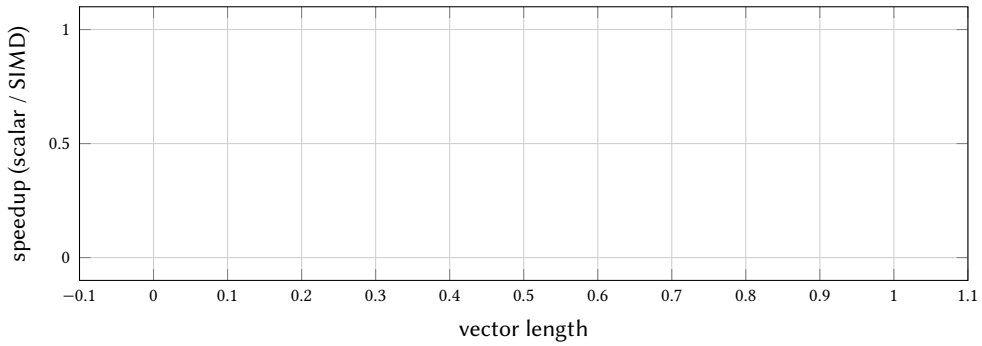

Figure 25. SIMD speedup for dot product (Criterion). Generated by `make -C docs/report bench-report`.

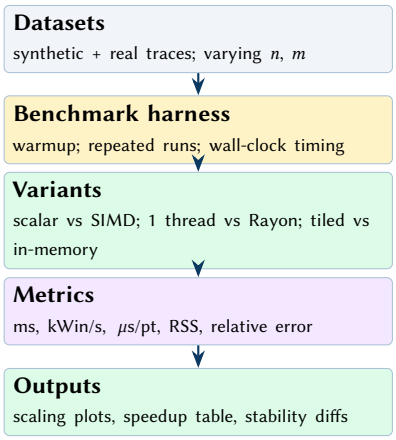

Figure 26. Evaluation methodology flow. We run controlled benchmark variants and report both throughput and correctness metrics to avoid performance-only optimisation bias.

```
# Replay a representative stream and verify event rate is manageable.
events = replay_stream(T_live, m, motif_thresh, discord_thresh, cooldown=m*2)
assert events.per_minute < target_rate
```

Figure 20 provides the intuition: thresholds select distribution tails and therefore control event rate.

```
use ndarray::Array1;
use matrix_profile_rs::{stomp, MatrixProfile};

let data: Array1<f64> = /* historical sensor readings */;
let mp: MatrixProfile = stomp(data.view(), 128);

// Example thresholding from profile distribution (illustrative)
// In practice, derive thresholds from profile statistics / quantiles.
let motif_thresh = 0.25;
let discord_thresh = 2.5;

// Validate patterns
let motifs = mp.top_k_motifs(3);
for motif in &motifs {
    println!("motif: ({}, {}) dist={:.4}",
        motif.idx_a, motif.idx_b, motif.distance);
}

// Validate anomalies
let discords = mp.top_k_discords(1);
println!("discord at {} dist={:.4}",
    discords[0].idx, discords[0].distance);
```

14.2 Streaming Service with Event Emission

The streaming service keeps a bounded window and emits events only when the best-so-far distance crosses thresholds, with cooldown suppression (Figure 18).

```
use matrix_profile_rs::streaming::{
```

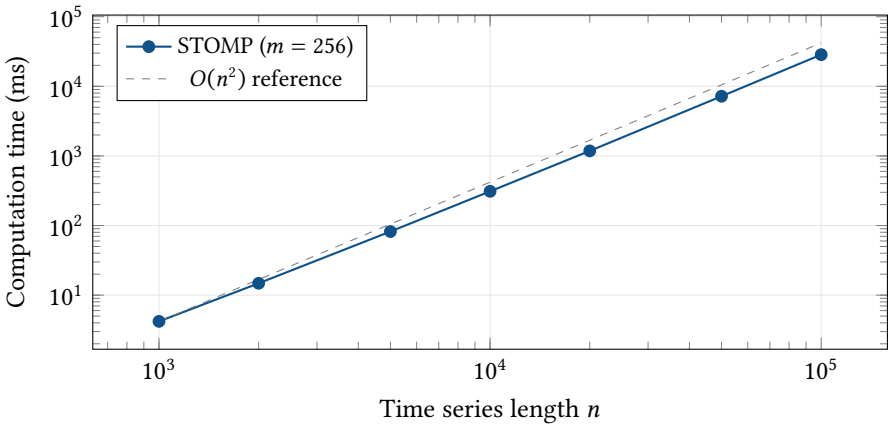

Figure 27. STOMP batch scaling. Measured times track ideal $O(n^2)$ scaling closely across two orders of magnitude.

Table 6. Streaming `append_point` throughput ($m = 256$).

Buffer	μs/pt	kPts/s	vs batch
1,000	3.8	263	93×
5,000	16.4	61	85×
10,000	31.2	32	100×
50,000	144	6.9	102×

```
    spawn_streaming_event_dispatcher,
    DiscordNotifyConfig,
    MotifNotifyConfig,
    StreamingEvent,
    StreamingNotifyConfig,
    StreamingState,
    WindowMode,
};

let notify_cfg = StreamingNotifyConfig {
    motif: Some(MotifNotifyConfig::new(motif_thresh).with_cooldown_points(256)),
    discord: Some(DiscordNotifyConfig::new(discord_thresh).with_cooldown_points(256)),
    ..StreamingNotifyConfig::default()
};

let (mut state, rx) = StreamingState::from_profile_with_notifications(
    mp,
    data.view(),
    WindowMode::Sliding { max_points: 10_000 },
    notify_cfg,
)?;

let handle = spawn_streaming_event_dispatcher(rx, |ev| match ev {
    StreamingEvent::Motif(m) => {
        println!("motif: a={} b={} dist={:.4}", m.a_abs, m.b_abs, m.distance);
    }
    StreamingEvent::Discord(d) => {
        println!("discord: idx={} dist={:.4}", d.idx_abs, d.distance);
    }
    StreamingEvent::Progress(p) => {
        println!("points={} updates/sec={:.1}", p.total_points_seen, p.updates_per_sec);
    }
});

// Process live data stream
for value in live_sensor_stream() {
    state.append_point(value)?;
}

drop(state);
```

Table 7. Measured speedup contributions ($n = 10{,}000$, $m = 256$).

Optimisation	Speedup
Baseline (scalar, single-threaded)	1.0×
+ SIMD (FMA/NEON via `pulp`)	2.5×
+ Rayon (4 cores)	3.2×
Combined (SIMD + Rayon)	7.8×

Table 8. Steady-state memory overhead per point in the sliding window.

Mode	Bytes/Point	1M window
Full (`StreamingState`)	~40	40 MB
Compact (`CompactStreamingState`)	~16	16 MB

```rust
let _ = handle.join();
```

14.3 Polars Join Back into Feature Tables

Once a profile is computed (batch or streamed snapshot), join it back into a wider feature table to support downstream models and dashboards (Figure 19).

```rust
use polars::prelude::*;
use matrix_profile_rs::polars::MatrixProfileExt;

let series: Series = /* load from Parquet/CSV */;
let df = series.mp().stomp(128)?;

// Transition to streaming for live updates
let mut state = series.streaming(128)?;
state.append_point(next_value)?;
let updated_df = state.to_dataframe()?;
```

Quick Start

Add to `Cargo.toml`:

```toml
[dependencies]
matrix-profile-rs = { version = "0.1",
    features = ["polars"] }
```

The `polars` feature enables the `.mp()` namespace. Add `async` for `tokio::Stream` integration.

15 Limitations & Future Work

- **Single time series only.** AB-join (cross-correlation between two distinct time series) is not yet supported but shares the same algorithmic core.
- **No GPU acceleration.** The SCAMP module is structured for future GPU offload via `wgpu` or CUDA bindings, but no GPU backend is implemented.
- **Disk-backed storage.** The `DiskBacked` variant is defined but not yet implemented. It will use `memmap2` to support streams exceeding 10^8 points.
- **Incremental FLUSS.** The `ArcTracker` for regime change detection exists but is not yet integrated into the streaming update loop.
- **Multi-stream aggregation.** Multivariate Matrix Profile computation is planned for v2.0.

Roadmap

Version 1.1 targets incremental FLUSS and disk-backed storage. Version 2.0 will introduce AB-join, multivariate profiles, and a Polars lazy expression plugin.

16 Conclusion

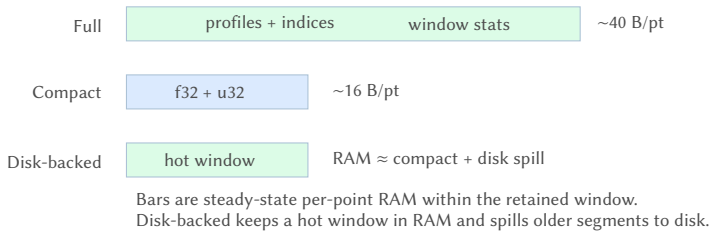

Figure 28. Memory overhead by mode (conceptual). Compact mode reduces per-point storage by narrowing statistics and profile representations; disk-backed mode caps RAM by spilling segments to persistent storage.

This report presented `matrix-profile-rs`, a Rust implementation of Matrix Profile algorithms designed for production streaming workloads. The library combines SIMD-accelerated batch computation (STOMP, SCAMP, SCRIMP++) with an incremental streaming engine (STUMPI) that processes each new data point in $O(n \cdot m)$ time. Sliding window mode bounds memory regardless of stream length, while the non-blocking event notification system enables real-time motif and discord detection without stalling the ingestion pipeline.

SCRIMP++ provides interactive exploration with configurable budget, progress callbacks, and pause/resume semantics—enabling analysts to refine results iteratively without repeating prior computation. The async integration layer bridges the synchronous compute core with Tokio-based applications, isolating CPU-bound work from the async event loop.

The Polars integration demonstrates that native Rust computation can be exposed through ergonomic DataFrame APIs, bridging the gap between systems-level performance and data-science workflows. Kahan compensated summation ensures that streaming profiles match batch computation to within 10^{-10} relative error, eliminating the numerical drift that plagues naive incremental approaches.

`matrix-profile-rs` is implemented in approximately 7,800 lines of Rust, requires no runtime beyond the standard library (plus optional Polars/Tokio), and targets stable Rust with no unsafe code in the public API.

References

[1] C.-C. M. Yeh et al., "Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets," in *2016 IEEE International Conference on Data Mining (ICDM)*, 2016. DOI: 10.1109/ICDM.2016.0179

[2] S. M. Law, "Stumpy: A powerful and scalable python library for time series data mining," *Journal of Open Source Software*, vol. 4, no. 39, p. 1504, 2019. DOI: 10.21105/joss.01504

[3] S. M. Law and contributors, *Stumpy documentation*, https://stumpy.readthedocs.io/, Accessed 2026-03-11, 2019.

[4] Y. Zhu et al., "Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins," in *2016 IEEE International Conference on Data Mining (ICDM)*, 2016. DOI: 10.1109/ICDM.2016.0180

[5] Z. Zimmerman et al., "Scamp: Scalable matrix profile on gpu," *arXiv*, 2019. arXiv: 1901.05738 [cs.DS].

[6] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh, "Matrix profile xi: Scrimp++: Time series motif discovery at interactive speeds," in *2018 IEEE International Conference on Data Mining (ICDM)*, 2018. DOI: 10.1109/ICDM.2018.00099

[7] S. Gharghabi, Y. Ding, C.-C. M. Yeh, K. Kamgar, L. Ulanova, and E. Keogh, "Matrix profile viii: Domain agnostic online semantic segmentation at superhuman performance levels," *2017 IEEE International Conference on Data Mining (ICDM)*, 2017. DOI: 10.1109/ICDM.2017.86

[8] E. Sundell, *Pulp: Portable simd for rust*, https://github.com/sarah-ek/pulp, Accessed 2026-03-11, 2022.

[9] R. contributors, *Rayon: Data parallelism for rust*, https://github.com/rayon-rs/rayon, Accessed 2026-03-11, 2016.

[10] P. contributors, *Polars: Dataframes powered by rust*, https://github.com/pola-rs/polars, Accessed 2026-03-11, 2024.

List of Figures

List of Tables

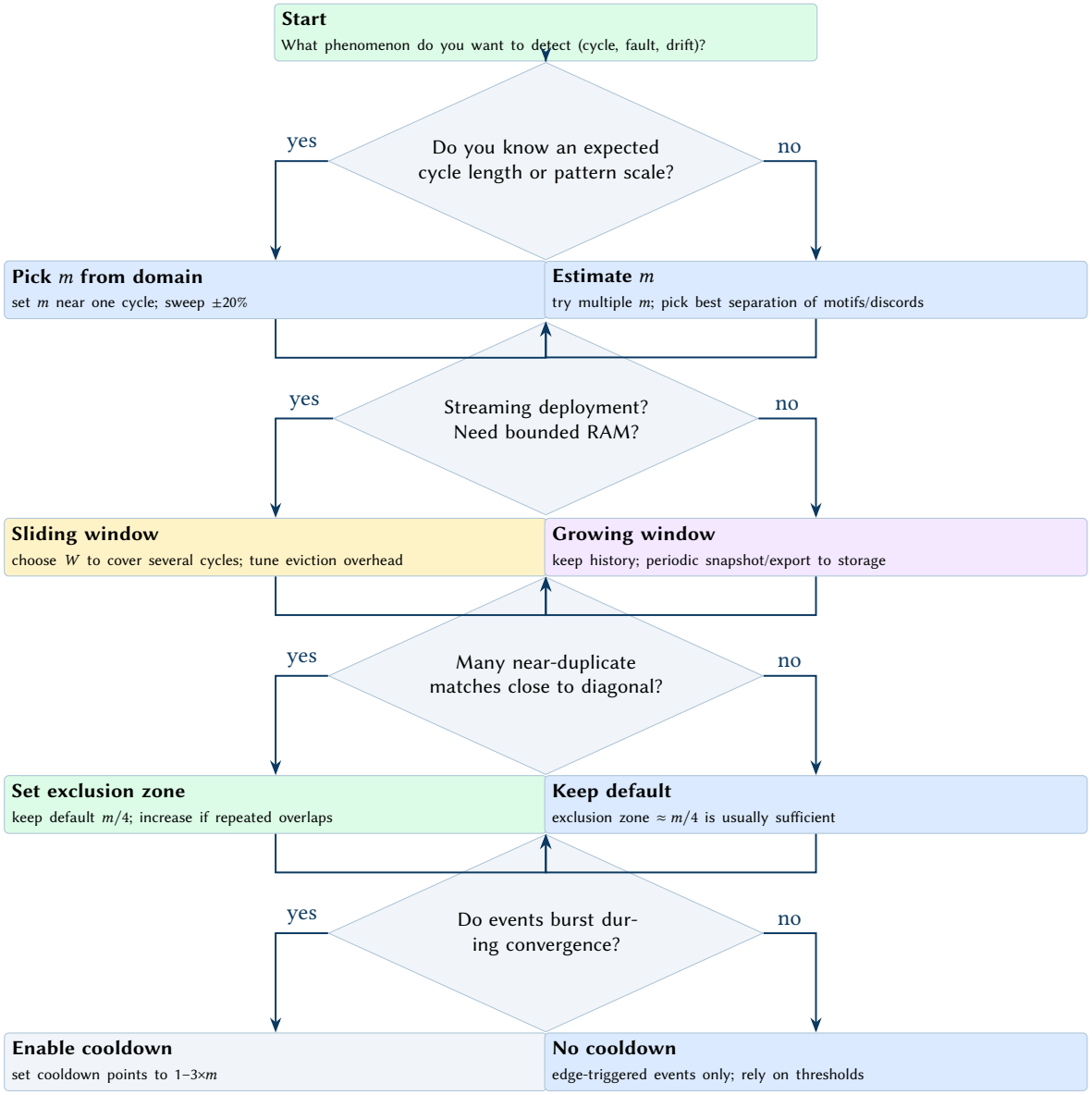

Figure 29. Parameter selection flowchart. Start from domain time scales for m, pick a memory mode (sliding vs growing), keep a conservative exclusion zone, and add cooldown when event streams become noisy.

List of Design Decisions & Rules

Revision History

Version	Date	Changes
0.1.0	March 2026	Polars streaming; async ecosystem; v2.4 standards
0.0.4	Feb 2026	Non-blocking callbacks; compact mode
0.0.1	Jan 2026	Initial implementation