Action-Scoped Percentile Telemetry for Frame Timing in Browser-Based Graphics Applications

Denis Saripov

***Abstract*—Interactive browser-based graphics applications are judged by smoothness, but production telemetry often reports averages that hide short stalls. This paper presents a low-overhead method for reporting frame timing percentiles in browser graphics applications. The method separates two metrics: frame interval, measured as the spacing between consecutive requestAnimationFrame callbacks, and CPU render duration, measured around the application render function. Samples are aggregated in fixed-duration or action-scoped windows and reported as compact percentile summaries instead of per-frame analytics events. A Canvas 2D whiteboard demo was instrumented with deterministic and browser-dependent stress injectors. In six repeated 5 s windows, a 50 ms stall before rendering raised both render-duration p95 and frame-interval p95 to 50.5 ms, whereas the same stall after rendering left render-duration p95 at 0.5 ms while frame-interval p95 rose to 50.5 ms. The results show that render duration alone misses user-visible main-thread stalls and that action-scoped percentile telemetry can identify whether a regression is inside or outside the render path.**

***Keywords*—browser graphics, Canvas, frame timing, jank, percentile telemetry, real-user monitoring, requestAnimationFrame.**

I. Introduction

Browser-based graphics applications now cover whiteboards, design tools, map viewers, scientific dashboards, and visualization systems. These applications compete with native interfaces on perceived smoothness. A single visible stall during zooming or dragging can matter more to a user than the average rendering cost across several seconds.

Mean frame time is therefore a weak production signal. A trace can average below the 16.7 ms budget of a 60 Hz display and still contain intermittent long frames. Tail latency has the same practical problem in large services: rare slow cases dominate experience at scale [1]. Graphics perception studies also show that temporal artifacts and frame-rate variation affect perceived quality [2], [3].

Existing browser and web-visualization work gives strong evidence that rendering technology, workload shape, and browser/runtime behavior all affect performance [4]–[12]. However, much production instrumentation still collapses frame behavior into a mean, a frames-per-second estimate, or a single render-function timer. That hides whether a visible stall came from the application render path, from other main-thread work, from garbage collection, or from browser scheduling.

This paper proposes a small telemetry pipeline for production browser graphics. It reports two percentile streams: frame interval I_k and CPU render duration R_k. It also introduces action-scoped windows: the current window is flushed before a user action, such as zooming, and the subsequent frames are tagged with that action. The goal is not to replace full tracing or GPU profiling. The goal is a cheap field signal that survives normal analytics constraints.

The contribution is threefold: first, a precise metric split between user-visible frame pacing and instrumented CPU render work; second, an action-scoped windowing strategy suitable for low-cardinality production analytics; third, a reproducible Canvas 2D benchmark showing that the split identifies stalls that render-only instrumentation misses.

II. Related Work

Web visualization performance has been studied across Canvas, SVG, WebGL, WebGPU, and domain specific systems. Khan et al. evaluated high-volume streaming visualization performance in web environments [4]. Schwab et al. translated interactive SVGs to a multithreaded virtual DOM to improve rendering scalability [5]. Zunino et al. compared web mapping libraries in a data visualization case study [6]. Herzberger et al. studied scalable browser-based volume rendering [7], while Sengupta et al. compared WebGL and WebGPU acceleration in modern browsers [8].

Several works focus on web animation and client-side rendering. Andrews and Wright used JavaScript and WebGL for browser-based information visualization [9]. Ren et al. proposed transparent GPU support for visualization rendering [10]. Beno and Olvecky measured dynamic 2D animation techniques in browsers [11], and Koren Ivancevic et al. benchmarked modern web animation systems across devices and operating systems [12]. A broader systematic review of website performance metrics also shows that metric choice and measurement method remain active research problems [13]. These works motivate the need for repeatable measurement, but they are mostly benchmark-oriented rather than production telemetry pipelines.

Browser standards expose several timing surfaces relevant to this problem. The High Resolution Time specification defines monotonic high-resolution timestamps [14]. The HTML Standard defines animation frame callbacks [15]. The Long Tasks API reports main-thread

Manuscript received April 25, 2026.
Denis Saripov – independent researcher (email: iam.saripov.denis@gmail.com).

tasks that take at least 50 ms. The Long Animation Frames (LoAF) API reports animation frames over 50 ms, including whole-frame duration, the browser-defined renderStart timestamp, blocking duration, and script details [16], [17]. The method in this paper overlaps with LoAF, but the measurements differ. LoAF uses browser-defined frame and rendering boundaries; R_k measures elapsed time across an application-defined main-thread render span, while I_k measures the spacing between animation callbacks. By contrast, the proposed method records every visible-tab callback, so missed refreshes below 50 ms—about 33.3 ms at 60 Hz—still appear in the summaries.

Percentile computation also matters. Greenwald and Khanna proposed space-efficient online quantile summaries [18]. DDSketch, t-digest, and stream-fusion approaches provide mergeable summaries useful for large-scale telemetry [19]–[21]. This paper uses 1 ms histograms for simplicity and auditability, while leaving mergeable quantile summaries as an implementation option for larger deployments.

III. Metrics and Windowing

For animation frame k, let $t_{\text{raf}}[k]$ be the timestamp supplied to requestAnimationFrame. The frame interval is $I_k = t_{\text{raf}}[k] - t_{\text{raf}}[k-1]$. It captures callback spacing and therefore includes render work, other main-thread tasks, browser scheduling, and missed display cadence. The first interval in a run is undefined and is excluded.

Let $t_{\text{start}}[k]$ and $t_{\text{end}}[k]$ bracket the application render function. CPU render duration is $R_k = t_{\text{end}}[k] - t_{\text{start}}[k]$. This measures the synchronous CPU work inside the application render span. It does not measure asynchronous GPU rasterization, compositor timing, or display scanout. The distinction is deliberate: R_k is an attribution signal, while I_k is closer to user-visible frame pacing.

Samples are aggregated in windows. A fixed-duration window is useful for idle animation or continuous rendering. An action-scoped window is useful for interaction analysis: the telemetry object flushes the current window before a user action, applies the action, then records the next window with an action label. Each closed window emits one summary event containing p50, p75, p90, p95, and p99 for R_k and I_k, slow-frame counts, workload settings, stress settings, and a small environment snapshot.

The action boundary is important because many graphics regressions are state-dependent. A map may be smooth while idle but slow immediately after a zoom level changes. A whiteboard may be smooth before selection but slow while a group of objects is dragged. If telemetry reports only arbitrary time windows, the regression is hard to attach to a user operation. If it flushes before the operation and labels the following window, the summary can be grouped by action, workload size, and browser environment without shipping a trace.

Per-frame analytics events are avoided. At 60 Hz, one user session can produce thousands of frames per minute; sending each frame to a third-party analytics system can create event-volume and self-interference problems. A window summary keeps cardinality low and makes the metric more suitable for field telemetry.

The percentile calculation in the reference implementation uses fixed 1 ms buckets. This is less sophisticated than a mergeable quantile sketch, but it has two advantages for production debugging: the error is easy to explain, and the bins can be merged later if raw bucket counts are retained. If a deployment needs accurate global p99 values across users, it should store histograms or a mergeable sketch rather than only storing the already-computed per-window percentiles.

IV. Algorithmic Specification

The online algorithm keeps two histograms and a few counters per active window. On each visible animation frame it receives the rAF timestamp, computes I_k from the previous rAF timestamp, measures R_k around the render function, adds both values to their histograms, and increments threshold counters for I_k values above 16.7 ms, 33.3 ms, and 50 ms. When the elapsed window duration reaches the configured limit, the window is closed and the histograms are converted into percentile estimates.

The action-scoped path uses the same data structure. Before an action runs, the current window is flushed so pre-action frames do not mix with post-action frames. The next window receives the action label. This avoids a common ambiguity in field telemetry: a bad five-second window might contain a zoom, a drag, and idle frames, making the cause unclear. Action scoping narrows the interpretation without requiring full tracing.

The algorithm has constant per-frame update cost with respect to the number of samples already collected. It does not sort frame samples, store raw per-frame arrays, or allocate per-frame telemetry objects for network submission. Memory grows with the number of histogram buckets and with the number of retained completed-window summaries, not with the number of frames in the current window. This is important for long sessions in editors, whiteboards, and dashboards.

The implementation treats telemetry as non-critical. If the sink fails, rendering continues. If a summary cannot be sent, the failure is local to telemetry and must not block the frame loop. This failure mode is part of the method: performance monitoring should not reduce the performance it is supposed to observe.

V. Reference Implementation

The reference application is a Canvas 2D whiteboard that renders 2,000 rectangular objects with optional animation, gradients, shadows, and text labels. It supports pan, zoom, and object dragging. A dedicated Zoom +10% control triggers an action-scoped measurement cycle.

The implementation records only visible-tab samples. If document.visibilityState is not visible, the frame is skipped to avoid background-tab throttling artifacts. The first completed window is discarded as warm-up. No outlier removal is applied. All visible foreground samples are recorded.

Three stress mechanisms are available. Busy-wait creates deterministic CPU stalls. Memory churn allocates and touches memory to increase garbage-collection pressure, a runtime behavior also studied in browser garbage-collection

scheduling work [22]. Canvas readback, used in Setup G before rendering, calls ctx.getImageData(0, 0, 512, 512) once every fifth visible requestAnimationFrame callback, inside the measured render span before the draw call. It reads the upper-left 512 × 512 region of the 1580 × 1080 canvas, not the whole canvas. The call returns an ImageData object containing a 1 MiB RGBA pixel buffer; the result is discarded. The context is created with alpha: false and without the willReadFrequently hint [23]. In these tests, memory churn and readback depended on the browser and hardware; only busy-wait was a deterministic tail generator.

The measured render span is intentionally narrow: it brackets the application render call, not the entire requestAnimationFrame callback. This makes R_k easier to interpret. Work placed before or after rendering, including layout code, analytics callbacks, event handlers, or third-party scripts, can still delay the next animation frame and therefore appear in I_k. This design gives a direct diagnostic question: did the tail come from the render path or from other main-thread work?

A closed window contains the number of observed frames, histogram summaries for R_k and I_k, counts above common frame-interval thresholds, the action label if present, and low-cardinality context. The context should include object count, mode flags, viewport size, device pixel ratio, and coarse browser family. It should not include high-cardinality values such as full URLs, raw user identifiers, or complete user-agent strings in a production analytics event unless the privacy model explicitly allows it.

Fig. 1. Canvas 2D whiteboard demo used for percentile instrumentation.

VI. Experiments

Measurements were collected in Chrome 147.0.7727.103 on macOS 26.3.1, Apple M4, 16 GB RAM. The browser viewport was fixed to 1920 x 1080 CSS pixels with devicePixelRatio = 1. The measured canvas area was 1580 x 1080 CSS pixels because the control sidebar occupied the remaining width. Developer tools were closed. Each setup used 5 s windows. One warm-up window was discarded, then six completed windows were recorded. Values in Table I are medians of per-window summaries; per-window percentiles were not merged.

Seven setups were used. Setups A-C represent normal interaction: a static 2,000-object scene, an animated scene with gradients, and the same animated scene immediately after a Zoom +10% action. Setups D and E inject the same 50 ms busy-wait every five frames, but at different points in the frame loop. Setup D injects before rendering; Setup E injects after rendering. Setups F and G test non-deterministic stressors: memory churn and canvas readback.

This design is deliberately narrow. It does not attempt to compare Canvas 2D against WebGL or WebGPU, because such comparisons are already covered by browser-rendering benchmark literature. Instead, the experiment asks whether the proposed telemetry can distinguish two operationally different cases that look similar to the user: a long frame caused by render work and a long frame caused by work outside the measured render span.

TABLE I. Frame timing summaries. Timing values are milliseconds.

Setup	Workload / stress	Windows	Frames / jank	R_k p95	I_k p95 / >33.3 ms
A	2k static, no jank	6	301 / 0	1.5	17.5 / 0.0%
B	2k animated + gradients	6	301 / 0	3.5	17.5 / 0.0%
C	Zoom +10%, animated + gradients	6	302 / 0	3.5	17.5 / 0.0%
D	50 ms busy-wait before render, every 5 frames	6	217 / 44	50.5	50.5 / 20.1%
E	50 ms busy-wait after render, every 5 frames	6	216 / 43	0.5	50.5 / 20.2%
F	128 MB memory churn before render, every 5 frames	6	301 / 60	3.5	17.5 / 0.0%
G	512 px canvas readback before render, every 5 frames	6	302 / 60	2.5	17.5 / 0.0%

VII. Discussion

The baseline, animated-gradient, and zoom-action workloads stayed close to one frame per refresh. Their p95 frame interval was 17.5 ms, and no interval above 33.3 ms was observed. This suggests that the measurement loop and action labels did not themselves create visible stalls in the tested workload.

The two busy-wait cases show the diagnostic value of the metric split. When the 50 ms stall was injected before rendering, it fell inside the measured render span, so both R_k

p95 and I_k p95 rose to 50.5 ms. When the same stall was injected after rendering, R_k p95 stayed at 0.5 ms while I_k p95 rose to 50.5 ms. A render-only metric would report the second case as healthy even though the user-visible frame interval was degraded.

Memory churn and canvas readback did not raise I_k p95 on this machine. Setup G raised R_k p95 from 1.5 ms in the static baseline to 2.5 ms, while I_k p95 remained 17.5 ms and no interval exceeded 33.3 ms. Thus, one 512 × 512 read every fifth callback did not create slow frame intervals on the tested browser and device. Larger readback regions and every-frame readbacks were outside the scope of this experiment.

The method remains application-side main-thread timing telemetry. It does not directly measure GPU time. It also cannot by itself attribute a long I_k value to a specific third-party script, layout task, or browser subsystem. Where supported [24], Long Task and LoAF entries can be collected through PerformanceObserver in the same reporting window as complementary root-cause signals. Long Task entries identify long main-thread tasks; LoAF entries provide browser-defined frame timing, blocking duration, and script details [16], [17].

The method is useful even when the application already tracks frames per second. FPS collapses a distribution into a rate. A frame stream with one 50 ms stall every five frames and a stream with uniform 20 ms frames can have similar average rates but different causes and different remediation paths. Percentiles and threshold counts preserve the tail shape well enough for production triage.

The metrics should not be used as a ranking score without context. A high R_k p95 can be acceptable for a heavy export preview but unacceptable during pointer dragging. A high I_k p95 during a hidden tab is an artifact, not a user-visible regression. A high p99 in a 5 s action window may represent only three samples. The telemetry event therefore needs enough context for filtering and enough sample count information for reviewers to judge stability.

VIII. Production Use

A production deployment can start with four rules. First, instrument both I_k and R_k. Second, discard hidden-document samples. Third, emit one summary event per window. Fourth, keep event dimensions low-cardinality. These rules avoid the common failure mode where performance telemetry itself becomes a source of jank or analytics cost.

The minimum payload is small: window duration, frame count, R_k percentiles, I_k percentiles, slow interval counts, action label, object count, rendering mode, viewport bucket, and device pixel ratio. For a custom telemetry backend, histogram bucket counts can also be submitted. For a generic product analytics backend, p50, p75, p95, and p99 values are usually easier to store, although they should not later be averaged as if they were raw samples.

Alerting should focus on regressions, not absolute thresholds alone. For example, a team can track I_k p95 for zoom and drag actions by workload bucket and browser family. If the zoom I_k p95 rises from 17.5 ms to 35 ms after a release while R_k p95 remains flat, the first investigation should be outside the measured render function: event handlers, state updates, layout, analytics callbacks, or other main-thread work. If R_k p95 rises with I_k p95, the render path is the first suspect.

Privacy and data-volume constraints shape the implementation. The event should avoid full user identifiers, full URLs, and unbounded action names. It should bucket continuous values such as object count or viewport size where exact values are unnecessary. If raw histograms are stored, retention should follow the same policy as other product telemetry.

IX. Event Schema and Aggregation

A practical event schema should be stable before the first production rollout. Metric names should separate render and interval streams, for example render_p50_ms, render_p95_ms, interval_p50_ms, and interval_p95_ms. Threshold counters should be stored as counts as well as rates when the backend allows it, because rates cannot be recomputed if the denominator is lost. The frame count is therefore not optional; it is required to interpret all percentiles and slow-frame rates.

Action labels should be short and enumerable: zoom_in, zoom_out, drag_selection, pan_canvas, open_file, or idle_window. They should not include object identifiers or document names. Workload fields should be coarse enough to keep cardinality bounded: object_count_bucket rather than object_count_exact, viewport bucket rather than exact window dimensions, and browser_family rather than a full user-agent string. Exact values are useful in lab experiments, but they can make production analytics expensive and hard to query.

For fleet-level reporting there are two acceptable aggregation policies. The approximate policy stores per-window percentiles and reports medians or distributions of those window summaries. This is cheap and often sufficient for release regression detection. The exact policy stores mergeable summaries such as histogram buckets, DDSketch, or t-digest state and computes global quantiles from the merged distribution. The invalid policy is to average p95 values and call the result a global p95.

The schema should also preserve the difference between sampling and reporting windows. A five-second window at 60 Hz usually contains about 300 frames, but a janky workload may contain fewer frames because long intervals reduce callback frequency. Reporting both duration and frame count exposes this effect. It also prevents a short partial window from being interpreted as if it had the same statistical weight as a full window.

X. Validity Threats

Timer precision and privacy mitigations can quantize high-resolution timestamps [14], [25]. The implementation reduces sensitivity to this issue by using 1 ms histogram buckets. Background tabs can throttle animation callbacks, so hidden-document frames are excluded. Operating-system noise can also distort browser benchmarks [26]. Developer tools can alter scheduling and were closed during collection.

The experiment used one browser, one operating system, and one device. The absolute values should not be

generalized as cross-device performance claims. The relevant result is structural: I_k and R_k diverge when a stall occurs outside the render span. Wider validation should repeat the runs on mobile devices, integrated and discrete GPUs, and multiple refresh rates.

Finally, p99 from short windows is noisy. At 60 Hz, a 5 s window contains about 300 frames, so p99 depends on only a few samples. This paper reports p99 for inspection but uses p95 and slow-frame rates as the primary operational signals.

XI. Conclusion

This paper presented a production-oriented telemetry method for browser graphics applications. The method separates requestAnimationFrame interval from CPU render duration, aggregates samples into fixed or action-scoped windows, and emits compact percentile summaries. The reference demo shows that a stall outside the render function can degrade frame interval while leaving render duration apparently healthy. Reporting both streams is therefore necessary for practical browser graphics telemetry.

The fastest path to deploy the method is to add the paired timers around the existing render loop, filter hidden-document frames, aggregate locally in 1 ms histograms, and submit one summary event per window. More advanced deployments can replace histograms with mergeable quantile sketches, but the core requirement remains the same: do not collapse frame pacing and render work into one mean.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013, doi: 10.1145/2408776.2408794.

[2] V. Hulusic, K. Debattista, and A. Chalmers, "Smoothness perception," *The Visual Computer*, vol. 29, no. 11, pp. 1159–1172, 2013, doi: 10.1007/s00371-012-0760-6.

[3] K. Debattista, K. Bugeja, S. Spina, T. Bashford-Rogers, and V. Hulusic, "Frame rate vs resolution: A subjective evaluation of spatiotemporal perceived quality under varying computational budgets," *Computer Graphics Forum*, vol. 37, no. 1, pp. 363–374, 2018, doi: 10.1111/cgf.13302.

[4] S. Khan, E. Rydow, S. Etemaditajbakhsh, K. Adamek, and W. Armour, "Web performance evaluation of high volume streaming data visualization," *IEEE Access*, vol. 11, pp. 15623–15636, 2023, doi: 10.1109/ACCESS.2023.3245043.

[5] M. Schwab, D. Saffo, N. Bond, S. Sinha, C. Dunne, J. Huang, J. Tompkin, and M. A. Borkin, "Scalable Scalable Vector Graphics: Automatic translation of interactive SVGs to a multithread VDOM for fast rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 9, pp. 3219–3234, 2022, doi: 10.1109/TVCG.2021.3059294.

[6] A. Zunino, G. Velazquez, J. C. Celemin, C. Mateos, M. Hirsch, and J. Rodriguez, "Evaluating the performance of three popular web mapping libraries: A case study using Argentina's Life Quality Index," *ISPRS International Journal of Geo-Information*, vol. 9, no. 10, art. 563, 2020, doi: 10.3390/ijgi9100563.

[7] L. Herzberger, M. Hadwiger, R. Kruger, P. Sorger, H. Pfister, E. Groller, and J. Beyer, "Residency Octree: A hybrid approach for scalable web-based multi-volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 1, pp. 1380–1390, Jan. 2024, doi: 10.1109/TVCG.2023.3327193.

[8] S. Sengupta, N. Wu, M. Varvello, K. Jana, S. Chen, and B. Han, "From WebGL to WebGPU: A reality check of browser-based GPU acceleration," in *Proc. 2025 ACM Internet Measurement Conference*, pp. 1018–1024, 2025, doi: 10.1145/3730567.3764504.

[9] K. Andrews and B. Wright, "FluidDiagrams: Web-based information visualisation using JavaScript and WebGL," in *Proc. Eurographics Conference on Visualization (EuroVis 2014)*, pp. 43–47, 2014, doi: 10.2312/eurovisshort.20141155.

[10] D. Ren, B. Lee, and T. Hollerer, "Stardust: Accessible and transparent GPU support for information visualization rendering," *Computer Graphics Forum*, vol. 36, no. 3, pp. 179–188, 2017, doi: 10.1111/cgf.13178.

[11] M. Beno and M. Olvecky, "Measuring the performance of techniques for dynamic 2D animation in web browsers," *Journal of Applied Mathematics, Statistics and Informatics*, vol. 20, no. 2, pp. 77–110, 2024, doi: 10.2478/jamsi-2024-0009.

[12] T. Koren Ivancevic, T. J. Jezic, and N. Stanic Loknar, "A cross-device and cross-OS benchmark of modern web animation systems," *Journal of Imaging*, vol. 12, no. 1, art. 45, 2026, doi: 10.3390/jimaging12010045.

[13] M. Ghattas, S. Odeh, and A. M. Mora, "Predicting website performance: A systematic review of metrics, methods, and research gaps (2010-2024)," *Computers*, vol. 14, no. 10, art. 446, 2025, doi: 10.3390/computers14100446.

[14] W3C, "High Resolution Time," Editor's Draft. [Online]. Available: https://w3c.github.io/hr-time/. Accessed: 25 Apr. 2026.

[15] WHATWG, "HTML Standard: Animation frames," Living Standard, updated 24 Apr. 2026. [Online]. Available: https://html.spec.whatwg.org/multipage/imagebitmap-and-animations.html#animation-frames. Accessed: 25 Apr. 2026.

[16] W3C, "Long Tasks API," Editor's Draft, 19 Mar. 2026. [Online]. Available: https://w3c.github.io/longtasks/. Accessed: 25 Apr. 2026.

[17] W3C, "Long Animation Frames API," Editor's Draft. [Online]. Available: https://w3c.github.io/long-animation-frames/. Accessed: 25 Apr. 2026.

[18] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 58–66, 2001, doi: 10.1145/375663.375670.

[19] C. Masson, J. E. Rim, and H. K. Lee, "DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2195–2205, 2019, doi: 10.14778/3352063.3352135.

[20] T. Dunning and O. Ertl, "Computing extremely accurate quantiles using t-digests," arXiv:1902.04023, 2019. [Online]. Available: https://arxiv.org/abs/1902.04023. Accessed: 25 Apr. 2026.

[21] M. Cafaro, C. Melle, I. Epicoco, and M. Pulimeno, "Data stream fusion for accurate quantile tracking and analysis," *Information Fusion*, vol. 89, pp. 155–165, 2023, doi: 10.1016/j.inffus.2022.08.005.

[22] U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer, "Idle time garbage collection scheduling," in *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 570–583, 2016, doi: 10.1145/2908080.2908106.

[23] WHATWG, "HTML Standard: The 2D rendering context, willReadFrequently," Living Standard. [Online]. Available: https://html.spec.whatwg.org/multipage/canvas.html#concept-canvas-will-read-frequently. Accessed: 3 Aug. 2026.

[24] MDN Web Docs, "PerformanceLongAnimationFrameTiming: Browser compatibility," Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/PerformanceLongAnimationFrameTiming. Accessed: 2 Aug. 2026.

[25] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Financial Cryptography and Data Security*, pp. 247–267, 2017, doi: 10.1007/978-3-319-70972-7_13.

[26] Z. Toufie and B. Kabaso, "OS noise mitigations for benchmarking web browser execution environment performance," *Discover Computing*, vol. 27, no. 1, Art. no. 37, 2024, doi: 10.1007/s10791-024-09471-4.