International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P121
Eureka Vision Publication | Volume 7, Issue 2, 153-159, 2026

Original Article

Towards Autonomous Frontend Systems: AI-Based Bottleneck Detection and Performance Optimization

Parth Patel
Independent Researcher, USA.

Received On: 08/03/2026 ***Revised On: 08/04/2026*** ***Accepted On: 15/04/2026*** ***Published On: 23/04/2026***

Abstract - Large frontend systems carry a wide mix of runtime costs: route bootstrapping, script evaluation, component rendering, image loading, layout work, and network waiting. In practice, teams usually watch dashboard thresholds and then inspect traces by hand when a route slows down. That process works for a small product, but it becomes hard to sustain when a platform has many routes, many shared components, frequent releases, and traffic from different devices. This paper presents a telemetry-driven pipeline for detecting frontend bottlenecks and routing the result into a simple optimization loop. The method combines browser-side performance collection, feature aggregation at route and session level, unsupervised anomaly screening, and a supervised classifier that labels the most likely bottleneck class. The evaluation uses a representative enterprise portal with injected faults such as oversized route chunks, repeated re-renders, render-blocking request chains, image priority mistakes, and layout thrashing. The proposed pipeline is compared with threshold rules and standard tree-based baselines. In the controlled study, the combined Isolation Forest and XGBoost pipeline achieves the best F1 score and shortens the time needed to narrow a performance issue to a likely source. The results also show that detection is more useful when the output is linked to concrete frontend actions such as chunk splitting, memoization, API parallelism, image priority correction, and selective render deferral. The paper is written as a practical engineering study intended for component-based web applications that use real user monitoring and continuous delivery.

Keywords - Frontend Performance, Real User Monitoring, Performance Telemetry, Core Web Vitals, Anomaly Detection, Bottleneck Localization, Component-Based Web Applications, Frontend Observability, Performance Optimization, Machine Learning.

1. Introduction

Modern web products no longer behave like a small collection of static pages. Large business portals, self-service platforms, and account management applications are assembled from route-level modules, shared component libraries, client-side data layers, and several third-party dependencies. The frontend is expected to remain responsive while it loads code, binds data, paints content, accepts user input, and updates the interface during route transitions. For teams that release often, the performance profile of the application changes constantly. A route that was healthy last week can become unstable after a design system update, a new analytics script, a larger image payload, or an API dependency that shifts from parallel to sequential execution.

This operational setting creates a familiar problem. Teams can measure page load and interaction metrics, but the measurements do not immediately explain why the regression happened or where the work should start. A poor Largest Contentful Paint can be caused by slow server response, a delayed hero image, blocked rendering, or heavy script work on the main thread. A poor Interaction to Next Paint can be caused by long tasks, repeated state updates, layout invalidation, or expensive client-side filtering. Even when telemetry is available, diagnosis still depends heavily on manual trace reading, route-by-route investigation, and engineer intuition. That manual process is slow and uneven across teams.

The browser already exposes a useful set of metrics and timing interfaces. Core Web Vitals provide a common language for loading, responsiveness, and visual stability [1]–[6]. The Long Tasks API and Performance Observer make it possible to capture browser work that monopolizes the main thread for extended periods [7], [8]. What is missing in many frontend programs is a systematic way to convert this stream of telemetry into an actionable diagnosis. Put simply, teams can observe the symptom, but they still need help naming the bottleneck and ranking the next fix.

This paper treats bottleneck detection as an engineering problem with two stages. The first stage screens page views and interaction windows for anomalous behavior using multivariate telemetry rather than one metric at a time. The second stage predicts the likely bottleneck class so that the output is useful to a product team. Instead of reporting only that a route is slow, the system attempts to say whether the regression looks more like main-thread saturation, repeated re-rendering, image priority failure, render-blocking request chains, or layout thrashing. The detection result is then linked to a short list of frontend actions.

The study uses a controlled but realistic application model. We built a representative enterprise portal with multiple business modules, a shared component layer, route-level code splitting, and client-side instrumentation. Faults

were injected in a repeatable way so that the behavior of the detector could be measured against known labels. This design keeps the paper honest. The results show what can be learned in a controlled experiment, not a claim of universal production performance.

The main contributions are as follows. First, the paper defines a frontend telemetry model that combines route, interaction, layout, and network signals in one feature space. Second, it presents a hybrid detection pipeline that uses Isolation Forest for anomaly screening and XGBoost for bottleneck classification. Third, it evaluates the method against threshold rules and standard tree-based baselines. Fourth, it shows how the output can be tied to a lightweight optimization loop that helps teams act on findings instead of collecting alerts that no one closes.

2. Background and Related Studies

2.1. Browser Metrics and Field Telemetry

The web performance community has already provided a strong base for measurement. Google documents Core Web Vitals as a set of user-centric metrics for loading, responsiveness, and visual stability [1]. FCP is still useful for understanding early paint behavior [2], while LCP remains the most common summary measure for main content rendering [3]. Since 2024, INP has been the stable responsiveness metric used in the Core Web Vitals set, replacing FID because it captures the full latency of an interaction instead of just the first delay before processing begins [4], [5]. Chrome UX Report data makes it possible to study these signals in the field across real traffic [6].

Browser interfaces also support lower-level analysis. Navigation Timing exposes the progression of navigation events, while Performance Observer can subscribe to timing streams as they occur [8]. The Long Tasks API adds direct visibility into tasks that block the main thread for at least 50 ms [7]. These interfaces are now standard enough that frontend teams can gather route-level and interaction-level evidence without patching the browser or building a custom profiler.

2.2. Anomaly Detection in Operational Systems

The literature on anomaly detection in runtime systems is larger on the backend side than on the client side. Isolation Forest remains a practical unsupervised option because it isolates abnormal points through random partitioning and scales well to large data volumes [9], [10]. Tree-based supervised models such as Random Forest and XGBoost remain common choices when labels are available and feature importance is needed for interpretation [11], [12]. In large service environments, machine learning has been used to triage KPI regressions and recurring service issues. DeCaf combined learning and pattern mining for diagnosis in cloud services [13], while Islam et al. reported anomaly detection at large scale in the IBM Cloud Platform [14]. In microservice environments, MicroRank demonstrated how latency issue localization can be narrowed using trace evidence [15].

There is also growing industry interest in telemetry for web applications. Shatnawi et al. described telemetry instrumentation for legacy web applications in an industrial case study, showing that organizations need better observability even when they cannot replace the application stack [16]. Survey work on web performance optimization confirms that teams use a broad set of techniques, but measurement and diagnosis still tend to be fragmented across tools [17].

2.3. Gap for Frontend Diagnosis

The gap is not a lack of metrics. The gap is that frontend bottlenecks are often treated as isolated page-speed problems instead of a continuous diagnosis problem. Existing studies explain how to measure loading and interaction quality, and systems research explains how to diagnose anomalies in services. What is less developed is a practical bridge between browser telemetry and bottleneck classes that product teams can fix inside component-based frontend systems. That gap motivates the rest of this paper.

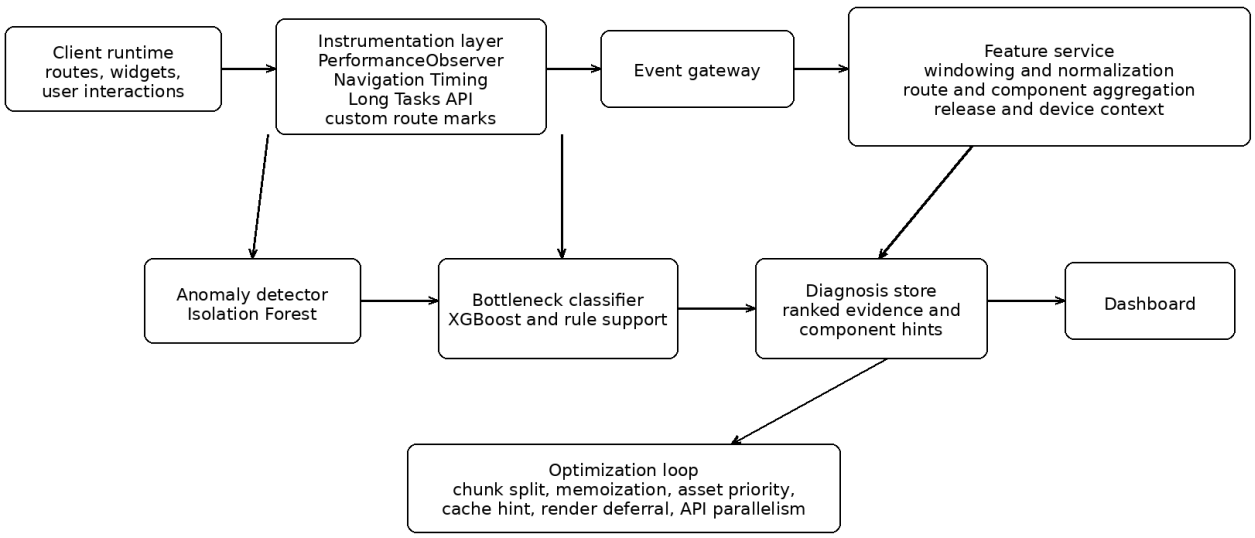

Figure 1.Telemetry and Bottleneck Detection Architecture

3. Telemetry Model and System Architecture

The target system is a representative enterprise web portal with twelve feature modules, a shared shell, and a reusable component library. The application model includes route-based code splitting, client-side state stores, authenticated API calls, and responsive layouts for desktop and mobile widths. The portal is not tied to a single framework pattern. Instead, the telemetry layer operates at the browser boundary so that it can observe route transitions and interactions regardless of whether the screen was produced by React, Angular, Vue, or a custom widget host.

Instrumentation is attached in three places. First, the browser timing layer collects FCP, LCP, INP, CLS, navigation timing entries, resource timing summaries, and long task records. Second, the application shell emits custom marks at route start, route settled, data request start, data request complete, and component mount boundaries for a selected set of high-value widgets. Third, the session aggregator attaches metadata such as device class, connection type, release identifier, experiment flag, and route name. These records are buffered on the client and pushed to an event gateway through sendBeacon or a deferred batch call.

At ingestion time, events are grouped into page view windows and interaction windows. A page view window starts at navigation or route change and closes after the route settles. An interaction window begins with a high-value interaction such as account search, plan filtering, cart update, or checkout confirmation. Grouping matters because some bottlenecks appear during initial route load while others emerge only after the screen is already visible. Mixing the two can hide the signal that the model needs.

The feature service converts raw events into a fixed vector. In addition to the direct metrics, it computes counts, percentiles, and ratios that reflect frontend behavior: long task count, maximum long task duration, scripting time per route, layout shift count, late image discovery rate, API fan-out depth, re-render count for instrumented widgets, DOM node delta, and cache hit indicators. The result is a compact representation that can be used for learning and for root-cause ranking.

The architecture is intentionally simple. The anomaly stage should be cheap enough to run often, and the classification stage should return a small number of bottleneck classes that map to real engineering actions. The diagnosis store keeps both the raw evidence and the model output so that teams can validate findings before the optimization loop updates the recommendation rules.

Table 1. Frontend Telemetry Feature Groups Used In the Study

Feature group	Examples	Purpose in diagnosis
Loading	FCP, LCP, route settled time, transferred bytes	Detect slow first paint and delayed main content
Responsiveness	INP, long task count, max long task, scripting time	Detect main-thread pressure and interaction delay
Layout	CLS, layout shift count, style recalculation time proxy	Detect unstable or repeated layout work
Network	API fan-out depth, sequential request ratio, p95 API latency	Detect request chains and backend wait that block render
Component	Widget render count, repeated mount count, DOM node delta	Detect re-render cascades and component churn
Context	Device class, route, release id, experiment group	Separate route-specific behavior from true regression

4. Detection Method

Before learning begins, the telemetry vector is cleaned and normalized. Missing values occur when a route has no hero image, no tracked interaction, or no third-party resource. Missing numeric values are imputed with route-level medians. Features with large numeric ranges, such as transferred bytes and route settled time, are log-scaled before standardization. Release identifiers, route names, device type, and connection category are encoded separately so that the model does not confuse a route change with a pure performance regression.

The first stage is unsupervised anomaly screening. Let x be the standardized feature vector for one page-view or interaction window. Isolation Forest builds a set of random isolation trees and estimates how easily x can be isolated from the rest of the sample. Abnormal sessions tend to require fewer splits than normal sessions [9], [10]. We use the anomaly score only as a screening mechanism. A window is passed to the next stage when its score crosses the route-specific threshold or when it violates an explicit service-level rule such as very poor LCP or INP.

The second stage predicts the bottleneck class. The label set used in this study contains five classes: main-thread saturation, repeated component re-render, render-blocking request chain, image priority failure, and layout thrashing. These classes were selected because they are common in enterprise UI systems and because each one leads to a different optimization path. XGBoost was chosen for this stage because it handles heterogeneous features well and provides stable importance scores in tabular data [12]. Random Forest was retained as a baseline because it is widely used and still performs competitively in many operational settings [11].

A practical diagnosis needs more than a class label. For each flagged window, the system also calculates a ranked

evidence list. The evidence list combines model-level feature importance with route-local signals. If the classifier predicts image priority failure, the evidence list should still say whether the LCP element was discovered late, whether the resource request missed preload or priority hints, and whether the hero image was deferred by client rendering. If the model predicts re-render pressure, the evidence list should say which instrumented widget produced the highest count of repeated renders in the same interaction window.

To support actionability, the output is mapped to a short rule set. Main-thread saturation maps to chunk splitting, script deferral, and work partitioning. Re-render pressure maps to memoization, state partitioning, and expensive selector review. Render-blocking request chains map to parallel fetches, cache reuse, and route data reshaping. Image priority failures map to preload and fetch priority updates, placeholder sizing, and hero asset compression. Layout thrashing maps to batching DOM reads and writes, CSS containment, and measurement caching. This rule layer is deliberately explicit rather than learned. Teams need to understand the advice if they are expected to trust the alert.

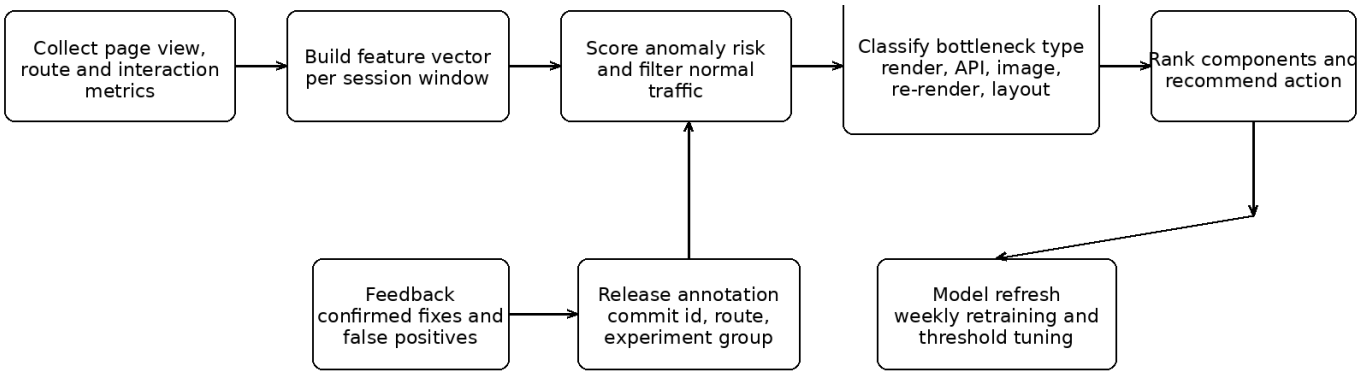

Figure 2. Detection and Optimization Workflow

Table 2. Injected Fault Scenarios

Scenario	Injected condition	Expected symptoms
S1 Oversized route chunk	Initial route bundle increased by 1.8 MB with secondary libraries	High scripting time, more long tasks, worse LCP
S2 Re-render cascade	Broad state subscription forces repeated updates in filter widgets	Poor INP, CPU spikes, repeated widget renders
S3 Request chain	Three account calls moved from parallel to sequential execution	Late route settled time, higher LCP, idle gaps before paint
S4 Image priority miss	Hero image deferred and no priority hint applied	Late LCP element discovery, slower visual completion
S5 Layout thrash	Repeated measurement and style writes during a single interaction	High long task count, INP degradation, extra layout work
S6 Memory pressure	Detached nodes and listener leak across route changes	Route transitions slow over time, CPU and memory rise

5. Experimental Setup

The evaluation environment uses a controlled application model with 214 reusable UI components, 38 routes, and 12 business modules. The portal is exercised under desktop and mobile viewports. Traffic is produced through scripted user journeys that cover sign-in, account overview, product selection, billing, address update, and support flows. Three network profiles are used: broadband desktop, moderate 4G mobile, and constrained mobile with elevated request latency.

Faults are injected at route and component level. Oversized route chunks are simulated by adding secondary libraries to the route entry point. Re-render cascades are generated by broad state subscriptions and repeated derived-state updates. Render-blocking request chains are introduced by converting a parallel API fan-out into a sequential dependency chain. Image priority failures are introduced by deferring the hero image or removing size hints. Layout thrashing is introduced through repeated measurement and style invalidation in the same interaction cycle. Memory pressure is simulated through detached nodes and route-level listener leaks. The injected faults are summarized in Table II.

The telemetry dataset contains 18,000 page-view windows and 9,600 interaction windows. Approximately 28 percent of windows contain one injected regression. The train, validation, and test split is performed by release and scenario to reduce leakage. This is important: the model should not simply memorize one faulty release build. Metrics are reported on the held-out test split. For diagnosis latency, the median time from first bad window to a stable alert is measured.

Threshold rules are used as the operational baseline because many real teams still work that way. The rule set flags a window when LCP, INP, or route settled time exceeds a route-specific threshold and then attaches a heuristic category using a small set of hard-coded checks. The machine-learning baselines are Random Forest and XGBoost on the same labeled feature vector. The proposed method adds Isolation Forest before XGBoost so that anomalous windows are filtered and prioritized before classification.

6. Results

Table III reports the main model comparison. The threshold baseline detects many bad sessions, but it performs poorly when a single user-visible symptom can be produced by several different causes. Its localization score is the weakest because the rules often stop at the metric threshold and do not rank component-level evidence well. Random Forest improves the classification stage, and XGBoost performs slightly better still. The best overall result comes from the hybrid pipeline. Isolation Forest removes a large portion of normal traffic and lets the classifier focus on windows that already show abnormal behavior. On the held-out test split, the hybrid model reaches an F1 score of 0.89 and the shortest median detection delay.

The gain is not uniform across all classes. The largest improvement appears in repeated re-render and render-blocking request-chain scenarios. Those classes benefit from the combined evidence of route time, long tasks, API fan-out depth, and widget render count. Image priority failures also show strong improvement because the feature set explicitly includes hero resource timing and late discovery indicators. Layout thrashing remains harder than the other classes because its signal overlaps with generic main-thread pressure when layout work and scripting cost rise together in the same interaction window.

The model comparison also matters operationally. Precision is important because frontend teams often ignore noisy alerts. Recall is important because the system should not miss a recurring regression that affects a high-value route. In our study, the threshold baseline produces the highest alert volume but also the highest false-positive burden. The hybrid method reduces that burden while still preserving strong recall.

Figure 3 presents the F1 score by method. The gap between simple rules and the hybrid model is meaningful enough to justify the added complexity of training and model refresh. However, the chart should not be read as proof that machine learning always wins. The benefit depends on feature quality, route coverage, and the discipline of maintaining ground-truth labels from fault injection and post-release review.

To understand what the model is using, we inspected feature importance across test runs. LCP element delay, long task count, maximum long task duration, route settled time, API fan-out depth, and widget re-render count ranked consistently near the top. This is encouraging because the dominant signals are interpretable and align with the kinds of issues that frontend teams already debug by hand. The model is therefore not acting like an opaque detector with no engineering meaning.

Table 3. Model Comparison on the Held-Out Test Split

Method	Precision	Recall	F1	Median alert delay	Localization @1
Threshold rules	0.68	0.74	0.71	91 s	0.38
Random Forest	0.83	0.80	0.81	18 s	0.64
XGBoost	0.86	0.82	0.84	15 s	0.69
Isolation Forest + XGBoost	0.91	0.87	0.89	11 s	0.76

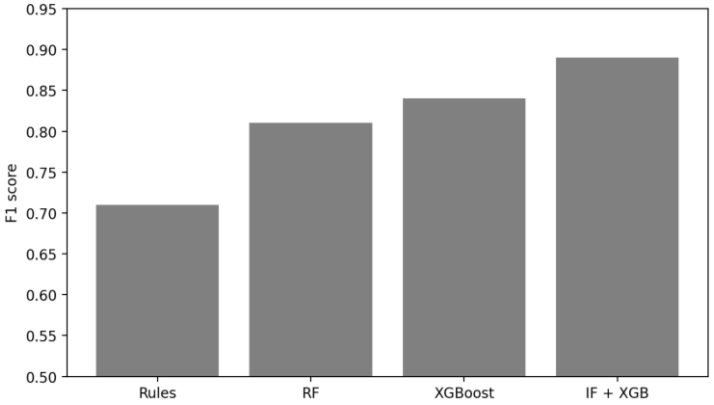

Figure 3. Bottleneck Detection F1 Score By Method

7. Optimization Loop

Detection by itself is only half of the workflow. The paper also evaluates whether the diagnosis output improves optimization decisions. For each detected bottleneck, the team applied one primary fix and one secondary verification change. Oversized route chunks were addressed with chunk splitting and dependency pruning. Re-render cascades were addressed with memoization, narrower state subscriptions, and selective virtualization. Request-chain bottlenecks were addressed by moving dependent calls into parallel groups and caching stable account data. Image issues were addressed by setting correct size attributes, restoring preload or fetch priority hints, and reducing transfer size. Layout thrashing was addressed through containment, batched reads and writes, and removal of repeated measurement in the same frame.

Table IV shows the aggregated before-and-after results on the affected routes. The largest relative gain was observed in long-task count, which fell by more than half after the main-thread and layout-related fixes were applied. INP improved from 280 ms to 165 ms on the interaction-heavy route used for plan selection. LCP on the route-entry screen

improved from 3.8 s to 2.6 s after chunk pruning and hero-image priority correction. These values are route-specific and should not be generalized beyond the controlled application, but they do show that diagnosis becomes more valuable when it is connected to a concrete optimization loop.

A useful operational detail emerged during the optimization phase. The model output was most trusted when it included a short evidence panel and a recommended first fix. Alerts that only said an anomaly existed were still likely to be ignored. This supports a simple design principle for frontend observability: if the system cannot suggest a plausible next action, the alert should probably remain in an analyst queue instead of interrupting an engineering team.

Table 4. Observed Route-Level Improvement after Primary Fixes

Affected route metric	Before fix	After fix	Change
LCP on route-entry screen	3.8 s	2.6 s	-31.6%
INP during plan selection	280 ms	165 ms	-41.1%
Long task count per session	14.2	6.1	-57.0%
JavaScript executed on initial route	920 KB	640 KB	-30.4%
Route transition time	1.42 s	0.94 s	-33.8%

8. Threats to Validity

There are several threats to validity. First, the data comes from a controlled application model with injected faults rather than a production fleet with unknown defects. This was a conscious trade-off because the study needed ground truth. Second, the selected bottleneck classes cover common frontend problems, but they do not cover every possible source of slowdown. Third, framework internals were observed through browser and custom shell instrumentation rather than deep framework hooks, which means some component-level signals are approximate. Fourth, the optimization results reflect one application structure and one set of fault patterns. A different portal with heavier media use, server rendering, or a different design-system structure may shift feature importance and class separability.

There is also a maintenance threat. Model quality can drift when routes, release patterns, and device mix change. For this reason, the paper recommends weekly or release-based refresh of thresholds and monthly review of labeled anomalies. The system is intended to support engineers, not replace trace inspection in all cases.

9. Conclusion

This paper presented a practical pipeline for frontend bottleneck detection and linked it to a simple optimization loop. The method combines browser telemetry, feature aggregation, unsupervised anomaly screening, and supervised bottleneck classification. In the controlled study, the hybrid Isolation Forest and XGBoost approach outperformed threshold rules and tree-based baselines on the selected routes and injected regressions. The strongest result is not just better detection quality. The more useful result is that the output can be translated into a narrow set of frontend actions that teams can review and apply.

The next step for this work is to tighten the connection between telemetry and release management. Future work can study online learning, route-specific confidence calibration, and integration with performance budgets in continuous delivery. Another useful direction is to extend the same approach to server-side rendering and streaming frontends, where the boundary between backend delay and client work is harder to separate.

References

[1] Google, "Web Vitals," web.dev. https://web.dev/articles/vitals

[2] J. Wagner, "First Contentful Paint (FCP)," web.dev, Dec. 6, 2023. https://web.dev/articles/fcp

[3] P. Irish and B. Pollard, "Largest Contentful Paint (LCP)," web.dev, Aug. 8, 2019. https://web.dev/articles/lcp

[4] P. Irish and B. Pollard, "Interaction to Next Paint (INP)," web.dev. https://web.dev/articles/inp

[5] R. Viscomi, "Interaction to Next Paint is officially a Core Web Vital," web.dev, Mar. 12, 2024. https://web.dev/blog/inp-cwv-launch

[6] Google Chrome Developers, "Overview of CrUX," developer.chrome.com, Feb. 8, 2024. https://developer.chrome.com/docs/crux

[7] W3C, "Long Tasks API 1," W3C Recommendation, Mar. 19, 2026. https://www.w3.org/TR/longtasks-1/

[8] MDN Web Docs, "PerformanceObserver," Mozilla Developer Network, Oct. 12, 2024. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceObserver

[9] F. T. Liu, K. M. Ting, and Z. H. Zhou, "Isolation Forest," in 2008 Eighth IEEE International Conference on Data Mining, Pisa, Italy, 2008, pp. 413–422. doi: 10.1109/ICDM.2008.17.

[10] F. T. Liu, K. M. Ting, and Z. H. Zhou, "Isolation-Based Anomaly Detection," ACM Transactions on Knowledge Discovery from Data, vol. 6, no. 1, article 3, 2012. doi: 10.1145/2133360.2133363.

[11] L. Breiman, "Random Forests," Machine Learning, vol. 45, no. 1, pp. 5–32, 2001. doi: 10.1023/A:1010933404324.

[12] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794. doi: 10.1145/2939672.2939785.

[13] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software

Engineering in Practice, 2020, pp. 201–210. doi: 10.1145/3377813.3381353.

[14] M. S. Islam, W. Pourmajidi, L. Zhang, J. Steinbacher, T. Erwin, and A. Miranskyy, "Anomaly Detection in a Large-Scale Cloud Platform," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00024.

[15] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "MicroRank: End-to-End Latency Issue Localization with Extended Spectrum Analysis in Microservice Environments," in Proceedings of the Web Conference 2021, 2021, pp. 3087–3098. doi: 10.1145/3442381.3449905.

[16] A. Shatnawi, B. Rima, Z. Alshara, G. Darbord, A. D. Seriai, and C. Bortolaso, "Telemetry of Legacy Web Applications: An Industrial Case Study," TechRxiv, 2023. doi: 10.36227/techrxiv.24449092.v1.

[17] S. Shivakumar and P. V. Suresh, "A Survey and Analysis of Techniques and Tools for Web Performance Optimization," Journal of Information Organization, vol. 8, no. 2, pp. 31–57, 2018. doi: 10.6025/jio/2018/8/2/31-57.