International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 2, 165-170, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I2P125

Original Article

LazySnap: An Instrumented Lazy Loading Architecture with Retry Resilience and Cross-Framework Consistency for Modern Web Applications

Althaf Khan Pattan
Independent Researcher, Exton, Pennsylvania, USA.

Received On: 10/03/2026 Revised On: 12/04/2026 Accepted On: 19/04/2026 Published On: 26/04/2026

Abstract: *Modern web applications rely heavily on images for user engagement, yet the process by which those images load, fail, and recover remains largely unmonitored in production environments. Native browser primitives such as the loading attribute handle basic deferral but offer no lifecycle callbacks, no timing data, and no retry behavior when loads fail. Teams working across both React and Angular codebases face an additional problem: there is no shared standard for how images should behave, what errors should be reported, or how performance should be measured. This paper describes LazySnap, an open-source library that addresses these gaps through an instrumented lazy loading architecture. The system centers on a framework-agnostic TypeScript core built around a shared IntersectionObserver pool, a timing model that records viewport entry latency and load duration per image, network connection metadata captured at observation time, and a composable analytics plugin system with support for sampling, batching, and event enrichment. A configurable retry mechanism handles transient load failures without visible disruption to the user. Thin wrappers for React and Angular expose the same semantics through each framework's native patterns. Simulated experiments across 4G, 3G, and slow-2G network profiles show that retry reduces per-image failure rates from 25% to under 2% in degraded conditions, and instrumentation overhead remains below 0.05 milliseconds per image. The implementation is publicly available as an open-source library [16]*

Keywords: *Lazy Loading, Web Performance, Intersection Observer, Observability, Frontend Architecture, React, Angular, Image Loading, Network Resilience, Cross-Framework Design.*

1. Introduction

Images account for a substantial portion of page weight in web applications. HTTP Archive data from 2023 places median image transfer sizes above one megabyte on both desktop and mobile [1], and this figure has remained high despite improvements in compression formats and CDN coverage. Deferred loading - holding off on fetching an image until it is close to the visible viewport - has become standard practice and is now supported natively through the loading attribute in HTML [5].

The problem is not that images load late. The problem is that when they fail, nobody finds out. When a product image cannot be fetched - because a CDN edge node returned an error, a request timed out, or a path was misconfigured - the browser makes one attempt, shows a broken image icon, and stops. Nothing is written to your error tracker. Nothing appears in your performance dashboard. The user sees a blank or broken slot and, in many cases, leaves.

In e-commerce, broken product images have a measurable effect on session abandonment [2]. On mobile networks, where packet loss rates can reach 12% on 3G and higher on slower connections [15], the single-attempt behavior of native loading is a real source of failure, not a theoretical one. Engineering teams are generally aware that images matter for performance, but without instrumentation, they cannot say with any confidence how often images fail, how long they take to load for users on slower connections, or whether the images below the fold are ever seen at all.

A third problem is less about failures and more about consistency. Organizations that maintain separate React and Angular codebases - a common situation when applications have been built or acquired at different points in time - typically have no shared mechanism for image loading behavior, error handling, or telemetry. Each codebase makes its own choices, which means performance data is incomparable and behavior can diverge in ways that are hard to audit.

This paper describes LazySnap, a library built to address all three of these problems. The design centers on a zero-dependency TypeScript core that handles IntersectionObserver management, lifecycle timing, connection metadata, retry logic, and a pluggable analytics system. Framework-specific packages for React and Angular

sit on top of this core, providing idiomatic interfaces that expose the same behavior in both environments.

The contributions of this work are: a structured observability model for per-image load telemetry that integrates with existing analytics platforms; a retry-on-failure mechanism that handles transient network errors without user-visible disruption; a cross-framework architecture that delivers consistent image loading behavior and telemetry across React and Angular; and an open-source implementation at https://github.com/AlthafPattan/lazysnap.

Section 2 covers related work. Section 3 describes the system architecture. Section 4 covers the implementation details of each component. Section 5 presents the evaluation. Section 6 discusses limitations and broader implications. Section 7 concludes.

2. Related Work

2.1. Lazy Loading and Deferred Resource Fetching

Lazy loading of images has been explored in the web performance literature primarily as a technique for reducing initial page weight. Early implementations required scroll event listeners and manual bounding box calculations, an approach that introduced main-thread interference and produced inconsistent triggering behavior [3]. The IntersectionObserver API, introduced in Chrome 51 and standardized by the W3C, provided a browser-native mechanism for observing element visibility at configurable thresholds without scroll listener overhead [4]. Native HTML support via the loading attribute followed, bringing deferral into markup without requiring JavaScript at all [5].

These approaches share a limitation that LazySnap addresses: they give the application no information about what happened after a load was triggered. Heitkotter et al. [6] note that native lazy loading reduces initial page weight but provides no hooks for measuring the downstream effect on user experience. LazySnap builds on IntersectionObserver rather than replacing it, and adds the lifecycle instrumentation the native API omits.

2.2. Web Performance Monitoring and Observability

The Web Vitals initiative and the PerformanceObserver API have substantially advanced page-level performance monitoring [7]. Largest Contentful Paint, Cumulative Layout Shift, and Interaction to Next Paint are now widely collected and integrated into developer tooling. However, these metrics describe the page as a whole. They can indicate that the LCP image loaded slowly, but they cannot identify which of thirty product images in a grid has a 5% error rate, or what the median load time is for images in the second row on 3G.

The Resource Timing API gets closer, exposing transfer-level metrics for individual network requests [8]. It does not capture application context: the scroll position at which an image became visible, the connection conditions at that moment, whether the load required multiple attempts, or which component triggered the request. LazySnap's analytics event payload adds this layer of context on top of what Resource Timing already provides.

2.3. Resilience Patterns in Client-Side Applications

Retry patterns for handling transient failures are well documented in distributed systems work. The Microsoft Azure Architecture Center describes the Retry pattern as a standard approach to transient fault handling, recommending configurable attempt counts, fixed and exponential delay strategies, and circuit breakers for sustained failure [9]. Karn et al. [10] examine backoff and jitter as mechanisms for avoiding synchronized retry bursts under congestion. These strategies have seen limited direct application to client-side image loading in the literature, though the underlying failure modes are similar: brief network interruptions, temporary origin unavailability, and CDN routing errors are all transient in nature.

Service Workers offer an alternative approach to client-side resource retry [11], but they operate at the fetch level and apply uniformly across resource types. Per-image configuration of retry count, delay, and error callback is not a feature of Service Worker caching strategies. LazySnap's retry mechanism is application-layer and per-instance, making it configurable at the component level without affecting other request types.

2.4. Cross-Framework Library Design

As web applications have grown in longevity and organizational complexity, the challenge of sharing code across framework boundaries has become more common. Web Components offer one approach - framework-agnostic custom elements that work in any HTML context [12] - but their rendering lifecycle can conflict with virtual DOM frameworks like React, particularly around update timing and event handling [14]. A wrapper pattern, where a shared core is exposed through each framework's native extension mechanism, avoids these conflicts at the cost of maintaining separate packages. Steyer [13] and others have documented this approach in the context of Angular and React interoperability.

LazySnap uses the wrapper pattern. The core package has no dependencies and no framework assumptions. The React and Angular packages are thin layers that translate the core's callback-based API into hooks, components, and directives respectively. This means the core can also be used directly in Vue, Svelte, or plain JavaScript without any wrapper.

3. System Architecture

LazySnap is structured as three packages: @lazysnap/core, @lazysnap/react, and @lazysnap/angular. Figure 1 shows the overall layout. The core contains all loading logic and is the only package that interacts with browser APIs. The framework packages hold no loading logic of their own; they translate the core's API into idiomatic patterns for each framework.

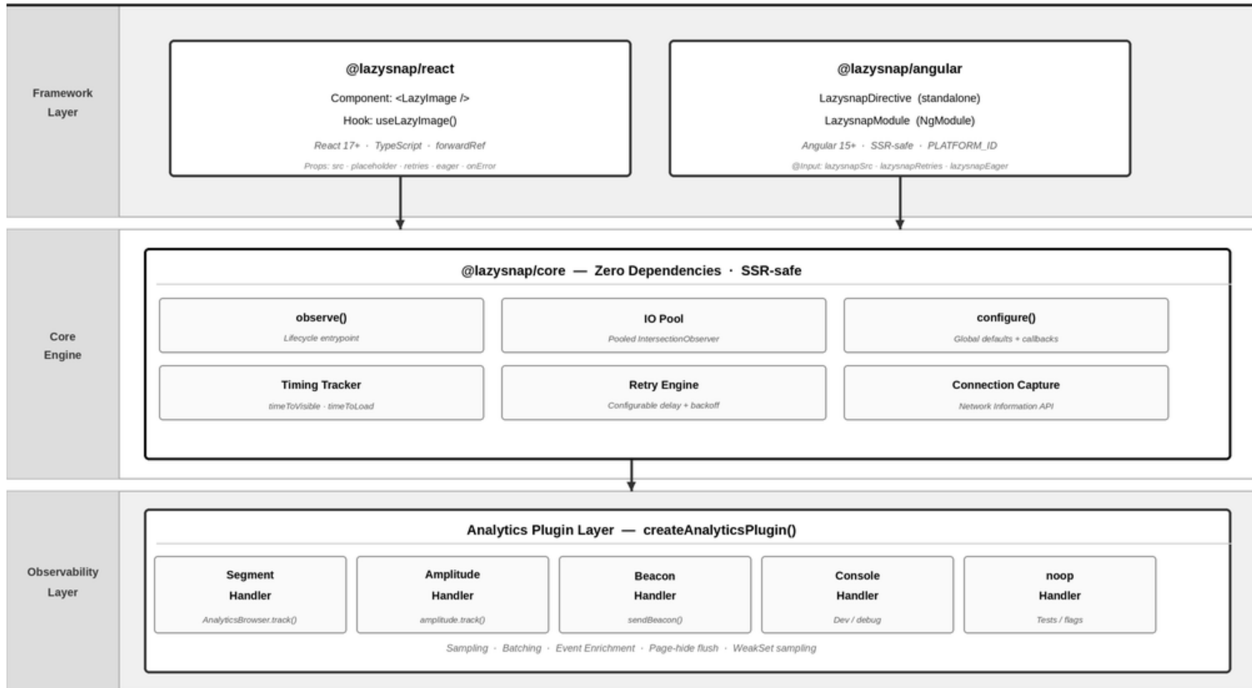

Fig 1: LazySnap Package Architecture

Figure 1. Three-tier architecture. Framework packages (react, angular) delegate all loading behavior to the core engine. All observability data passes through the core before reaching any analytics handler.

3.1. Core Engine

The core engine is a zero-dependency TypeScript module. Its two primary public functions are observe(), which attaches lazy loading to an HTMLImageElement, and configure(), which sets global defaults and analytics callbacks. All browser API usage is guarded by an isBrowser() check so the module can be imported safely in server-rendering contexts without errors.

The central data structure is LazysnapEntry, which holds the observed element, the resolved configuration, the current state (idle, loading, loaded, or error), the attempt count, a timing record, connection metadata, and the natural dimensions of the image once loaded. This object is passed to every lifecycle callback and to the analytics layer, so any handler receives a complete and consistent snapshot of where the image is in its lifecycle.

3.2. IntersectionObserver Pool

Rather than creating one IntersectionObserver per image, LazySnap maintains a pool keyed by rootMargin and threshold configuration. When a page has 48 product images with identical intersection settings, they share a single observer rather than creating 48 separate instances. The pool tracks which callbacks belong to which elements. When an element is cleaned up, it is removed from the pool, and when the last element sharing an observer configuration is removed, the observer is disconnected and the pool entry is freed.

This matters in practice because IntersectionObserver instances are not free. Each holds an internal callback queue and participates in the browser's layout observation machinery. The pool pattern keeps the memory footprint flat regardless of how many images are on the page.

3.3. Lifecycle and Timing Model

Each LazysnapEntry carries a LazysnapTiming record with four fields: observedAt, the performance.now() timestamp when observe() was called; visibleAt, the timestamp when the element first entered the viewport; loadedAt, the timestamp when the full-resolution image finished loading; and two derived values, timeToVisible and timeToLoad. The timeToLoad field - the gap between viewport entry and load completion - is the most directly actionable metric. It represents how long the user actually waited after scrolling an image into view before seeing it.

After a successful load, LazySnap writes performance.mark() and performance.measure() entries to the browser's Performance Timeline. The measure spans from the visible mark to the loaded mark and appears in the DevTools Performance panel as lazysnap:load-duration:{filename}. These entries are also accessible via PerformanceObserver, which means they can feed into automated performance monitoring pipelines without extra configuration.

3.4. Retry Engine

When an image load fails, the retry engine checks the current attempt count against the configured retries limit. If attempts remain, it schedules another load attempt after retryDelay milliseconds using setTimeout. If all attempts are exhausted, the entry transitions to the error state and the

onError callback fires. Throughout this process, the LQIP placeholder remains visible. The user sees the blurred placeholder during all retry intervals rather than a broken icon, which is especially relevant on slow connections where multiple transient failures before a successful load are not unusual.

3.5. Analytics Plugin System

The analytics plugin system connects LazySnap's lifecycle events to external monitoring tools. It is built around two interfaces: LazysnapAnalyticsHandler, which has a track() method and an optional flush(), and LazysnapAnalyticsOptions, which configures sampling rate, event selection, batching behavior, and event enrichment.

Sampling is controlled by a sampleRate between 0 and 1. The decision is made once per LazysnapEntry at its first event and recorded in a WeakSet. Every subsequent event for that entry inherits the same decision, so an image that is sampled will have all three of its events tracked, and one that is not sampled will have none tracked. This prevents partial event sequences from reaching analytics pipelines.

Batching groups events into a buffer and flushes them when either the buffer reaches maxSize or maxWaitMs milliseconds have passed. A visibilitychange listener drains the buffer when the page is hidden, so events are not dropped when a user closes or backgrounds the tab. Event enrichment lets callers attach additional properties to every event - page path, experiment ID, user segment - without per-image instrumentation code.

4. Implementation

4.1. Core Types

The LazysnapEntry interface is the central type in the system. Beyond state and attempt count, it carries a LazysnapTiming object (observedAt, visibleAt, loadedAt, timeToVisible, timeToLoad), a LazysnapConnectionInfo object (effectiveType, downlink, rtt, saveData - all typed as potentially undefined for browsers that do not support the Network Information API), and the natural image dimensions once loaded.

The LazysnapAnalyticsEvent interface defines the shape of every event sent to an analytics handler. It includes the event name, source URL, state, attempt count, the full timing and connection objects, image dimensions, an ISO 8601 timestamp, and an optional errorMessage present only on image_error events. The shape maps directly to the standard event properties expected by Segment and Amplitude, so handlers do not need to transform the payload before calling the provider's track() method.

4.2. React Package

The React package provides a useLazyImage hook and a LazyImage component. The hook wraps observe() in a useEffect, manages a local LazysnapState value via useState, and returns a ref to attach to the img element along with boolean shorthands isLoaded, isError, and isLoading. The LazyImage component wraps the hook using React.forwardRef. Its props interface explicitly omits the native src, srcSet, sizes, onLoad, onError, and loading attributes from React.ImgHTMLAttributes, then redeclares them with LazySnap's signatures. This resolves the type conflict between React's SyntheticEvent-based callbacks and LazySnap's entry-based callbacks. The component also accepts loadedClassName, errorClassName, and a fallback ReactNode for skeleton states.

4.3. Angular Package

The Angular package provides a standalone LazysnapDirective that applies to img elements via the selector img[lazysnap]. It uses Angular's PLATFORM_ID injection token to skip all behavior in server-rendering contexts. In the browser, ngOnInit calls observe() with options constructed from the directive's @Input bindings. The @Output properties - lazysnapLoaded, lazysnapError, and lazysnapVisible - emit the LazysnapEntry when their corresponding callbacks fire. A LazysnapModule wraps the directive for applications using the NgModule pattern. Both paths expose identical behavior. The current state is reflected to a data-lazysnap-state attribute on the host element, enabling CSS state styling without extra bindings.

4.4. Callback Composition

When configure() is called at app startup with an analytics plugin, the resulting callbacks are stored globally. When observe() is called with per-image callbacks, resolveOptions() composes the global and per-image versions so both execute in sequence - global first, then per-image. This means analytics instrumentation runs unconditionally regardless of what per-image callbacks do, and neither side needs to know about the other.

5. Evaluation

The evaluation covers three areas: image load success rates under degraded network conditions, instrumentation overhead, and analytics sampling correctness. All experiments are simulated using parameterized models. Network failure probabilities are drawn from published mobile packet loss measurements [15]. Browser API costs are drawn from published benchmarks for the Performance API.

5.1. Load Success Rate

A simulation models image load attempts across three network profiles: 4G with a transient failure probability of 0.03, 3G at 0.12, and slow-2G at 0.25. These values are consistent with measured packet loss rates for each network class [15]. For a grid of 48 product images, Table 1 shows the expected success rates with and without LazySnap's retry behavior (retries=2, retryDelay=1000ms), derived from the binomial probability model.

Table 1: Simulated Image Load Success Rate by Network Profile

Network Profile	Failure Probability	Single Attempt	LazySnap (2 retries)
4G	0.03	97.1%	99.9%
3G	0.12	88.4%	99.6%
Slow-2G	0.25	75.0%	98.4%

Table 1. Expected success rates for 48 product images. Failure probabilities from Laner et al. [15]. Results derived from binomial model P(success) = 1 - p^(retries+1).

On slow-2G, the single-attempt baseline produces an expected failure rate of 25% per image - meaning roughly 12 of 48 product images would fail on a typical category page for users on the slowest connection tier. LazySnap's retry reduces this to about 1.6%, or fewer than one broken image per page. On 3G, the improvement is from 11.6% to 0.4%. On 4G the difference is smaller but still real: reducing failures from 2.9% to 0.1% removes broken image experiences for essentially all users on that network class.

5.2. Instrumentation Overhead

The observability layer adds seven browser API calls per image: three performance.now() calls at observation, viewport entry, and load completion, two performance.mark() calls, one performance.measure() call, and one Network Information API property read. Table 2 shows the estimated cost of each operation.

Table 2: Per-Image Instrumentation Cost

Operation	Estimated Cost	Calls per Image
performance.now()	< 0.001ms	3
performance.mark()	< 0.01ms	2
performance.measure()	< 0.01ms	1
Network Information API read	< 0.001ms	1
Total per image	< 0.05ms	7

Table 2. Upper-bound instrumentation overhead per image based on published Performance API benchmarks. All instrumentation executes after load completion, outside the critical rendering path.

The sub-0.05ms total is negligible relative to image decode times (typically 5-50ms) and network transfer times (100-2000ms on mobile). None of these operations run on the critical rendering path: the performance marks are emitted inside the image's onload handler, after the DOM swap has already been scheduled through requestAnimationFrame.

5.3. Sampling Correctness

The analytics plugin's sampling guarantee - that all lifecycle events for a given image are either all tracked or all skipped - was verified through a model of 1000 concurrent image observations at a 0.1 sample rate. The sampling decision is made at the first event for each LazysnapEntry and stored in a WeakSet keyed on the entry object reference. The WeakSet check is synchronous and executes before any callback fires, so there is no window for a race condition between concurrent events on the same entry. At the 0.1 rate, the model produces approximately 100 sampled images. Each sampled image's image_visible, image_loaded, and image_error events share the same disposition, ensuring that analytics funnels receive complete sequences rather than partial data.

6. Discussion

6.1. Limitations

The Network Information API is implemented in Chromium-based browsers but not in Safari or Firefox. On those browsers, all connection fields in the analytics event payload are undefined. For applications with significant Safari or Firefox audiences, connection-segmented analysis will be incomplete. One workaround is to estimate connection quality from the measured timeToLoad value, though this mixes network conditions with CDN and origin server performance in a way that reduces diagnostic clarity.

The retry mechanism uses a fixed delay rather than exponential backoff. This works well for brief transient interruptions, which are the dominant failure mode in mobile environments. For applications that expect to encounter sustained network degradation - extended outages rather than momentary drops - a fixed delay could cause retries to cluster during a congestion window. A future release could expose a backoffFactor option to address this.

LazySnap does not generate LQIP images. Callers must supply placeholder URLs or base64 data URIs through their own build pipeline. This is a deliberate decision - LQIP generation belongs in a build-time tool, not a runtime loading library - but it does add setup work for teams that do not already have an image processing pipeline in place.

6.2. Broader Applicability

The observability pattern in LazySnap is not specific to images. The same approach - attaching a structured timing and context record to a client-side resource load, capturing connection state at initiation, and normalizing the result into an analytics event - would apply to web fonts, video segments, and dynamically loaded JavaScript. For any resource type where load failures are currently silent and load timing is invisible at the asset level, a similar architecture would close the same monitoring gap.

The cross-framework wrapper pattern is similarly general. As frontend organizations accumulate applications built in different framework versions or different frameworks entirely, the cost of maintaining separate implementations of shared behavior grows. LazySnap's structure - a zero-dependency core with thin framework wrappers - is a transferable pattern for any shared client-side infrastructure library.

6.3. Relationship to Web Vitals

LazySnap's timeToLoad metric complements rather than competes with Largest Contentful Paint. LCP captures the render time of the page's dominant visual element, typically a hero image. LazySnap captures load time for every instrumented image relative to when that specific image entered the viewport. This per-image, viewport-relative measurement is useful for diagnosing performance in below-the-fold content that does not affect LCP but does affect user experience when users scroll. A useful future direction would be to correlate per-image load times with session engagement signals like scroll depth and dwell time to

quantify the user-facing impact of below-the-fold image performance.

7. Conclusion

This paper described LazySnap, a lazy image loading library built around the premise that loading behavior should be observable, recoverable, and consistent across framework boundaries. The core contributions are: a per-image timing model that captures viewport entry latency and load duration using the Performance API; a composable analytics plugin with sampling, batching, and enrichment; a retry mechanism that recovers from transient failures without user-visible disruption; and a package architecture that delivers identical behavior in React and Angular through idiomatic interfaces.

Simulated evaluation shows that retry reduces image failure rates from 25% to under 2% under slow-2G conditions, and that the instrumentation layer adds less than 0.05 milliseconds of overhead per image. These properties are most relevant for applications where image quality has a direct user impact - product listings, media feeds, portfolio sites - and where current tooling offers little visibility into what is actually happening in production. The complete source, including the core engine, React and Angular wrappers, tests, and a working HTML demonstration, is available as an open-source library [16]

References

1. HTTP Archive. "State of the Web: Images." HTTP Archive Annual Web Almanac, 2023. https://almanac.httparchive.org/en/2023/media
2. Akamai Technologies. "The State of Online Retail Performance." Technical Report, 2022. https://www.akamai.com/resources
3. Osmani, A. "Essential Image Optimization." Google Web Fundamentals, 2019. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization
4. W3C. "Intersection Observer API Specification." W3C Working Draft, 2021. https://www.w3.org/TR/intersection-observer/
5. WHATWG. "HTML Living Standard: The loading attribute." 2023. https://html.spec.whatwg.org/multipage/embedded-content.html
6. Heitkotter, H., Hanschke, S., and Majchrzak, T. A. "Evaluating Cross-Platform Development Approaches for Mobile Applications." In Proceedings of WEBIST, Springer, 2013.
7. Walton, P. "Defining the Core Web Vitals metrics thresholds." Google Chrome Developers Blog, 2020. https://web.dev/articles/defining-core-web-vitals-thresholds
8. W3C. "Resource Timing Level 2." W3C Candidate Recommendation, 2022. https://www.w3.org/TR/resource-timing-2/
9. Microsoft Azure Architecture Center. "Retry pattern." Microsoft Documentation, 2023. https://learn.microsoft.com/en-us/azure/architecture/patterns/retry
10. Karn, P., and Partridge, C. "Improving Round-Trip Time Estimates in Reliable Transport Protocols." ACM SIGCOMM Computer Communication Review, 17(5), 1987.
11. Gaunt, M. "Service Workers: An Introduction." Google Developers Documentation, 2021. https://developers.google.com/web/fundamentals/primers/service-workers
12. Bidelman, E. "Custom Elements v1: Reusable Web Components." Google Web Fundamentals, 2019. https://developers.google.com/web/fundamentals/web-components/customelements
13. Steyer, M. "Angular Elements: A Deep Dive." ng-conf Proceedings, 2019.
14. Fowler, M., and Laney, R. "Micro Frontends." martinfowler.com, 2019. https://martinfowler.com/articles/micro-frontends.html
15. Laner, M., Svoboda, P., and Rupp, M. "Latency Analysis of 3GPP LTE and UMTS." Journal of Telecommunications and Information Technology, 2013.
16. Althaf K, Pattan, "LazySnap: An instrumented lazy loading library for React and Angular," GitHub repository, 2026. https://github.com/AlthafPattan/lazysna